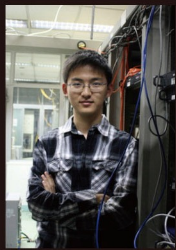
TURING 图灵原创



BYVoid 编著

Node.js 开发指南

人民邮电出版社
POSTS & TELECOM PRESS



BYVoid，清华大学计算机系2010级本科生，曾获得信息学奥林匹克竞赛（NOI）金牌。他从中学开始涉足开源开发，参与过“汉典网”等许多Web项目的前后端设计开发，同时是Linux输入法ibus-pinyin的作者。他从2009年Node.js诞生之始，就一直在关注它的发展，有许多使用Node.js建立网站的经验，活跃于CNode社区。

图书在版编目（C I P）数据

Node.js开发指南 / 郭家宝编著. -- 北京 : 人民邮电出版社, 2012.7
（图灵原创）
ISBN 978-7-115-28399-3

Ⅰ. ①N… Ⅱ. ①郭… Ⅲ. ①JAVA语言－程序设计－指南 Ⅳ. ①TP312-62

中国版本图书馆CIP数据核字(2012)第110717号

内 容 提 要

本书首先简要介绍Node.js，然后通过各种示例讲解Node.js的基本特性，再用案例式教学的方式讲述如何用Node.js进行Web开发，接着探讨一些Node.js进阶话题，最后展示如何将一个Node.js应用部署到生产环境中。

本书面向对Node.js感兴趣，但没有基础的读者，也可供已了解Node.js，并对Web前端/后端开发有一定经验，同时想尝试新技术的开发者参考。

图灵原创
Node.js开发指南

◆ 著　　BYVoid
责任编辑　王军花
执行编辑　丁晓昀

◆ 人民邮电出版社出版发行　北京市崇文区夕照寺街14号
邮编　100061　电子邮件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
北京　　　　印刷

◆ 开本：800×1000　1/16
印张：11.75
字数：249千字　　2012年7月第1版
印数：1－5 000册　　2012年7月北京第1次印刷

ISBN 978-7-115-28399-3

定价：45.00元

读者服务热线：(010)51095186转604　印装质量热线：(010)67129223

反盗版热线：(010)67171154

前　　言

这本书讲了什么

本书是一本 Node.js 的入门教程，写给想了解 Node.js 的开发人员。我的目标是使读者通过阅读本书，学会使用 Node.js 进行 Web 后端开发，同时能熟悉事件驱动的异步式编程风格，以便进一步了解 Node.js 的许多高级特性，以及它所应用的更多领域。

本书共6章，分别讨论了 Node.js的背景、安装和配置方法、基本特性、核心模块以及一些进阶话题。除此之外，还有2个附录，分别介绍了JavaScript 的高级特性和 Node.js 编程规范。下面简要概述各章的主要内容。

第 1 章“Node.js 简介”

这一章概述了什么是Node.js。读过这章后，你将对 Node.js 有一个基本的认识，同时了解它与 JavaScript 的深厚渊源。

第 2 章“安装和配置Node.js”

这一章讲述了如何在各种不同的环境下安装和配置 Node.js 及其基本运行环境，同时你你可以了解到如何编译Node.js，以及多版本管理工具。

第 3 章“Node.js 快速入门”

这一章讲解Node.js的基础知识，你将会学到如何使用 Node.js 的基本环境和工具进行开发、运行和调试。同时，还会讲解异步式I/O与事件式编程的一些重要概念，这些概念将会贯穿全书。此外这一章还详细介绍了 Node.js 的模块和包的系统，这些都是开发中经常会碰到的内容。

第 4 章“Node.js 核心模块”

这一章以全局对象、基本工具、事件发射器、文件系统和 HTTP 为代表，介绍了 Node.js

最常用的核心模块。你将会在后面的章节及以后的开发中经常与这些模块打交道。

第 5 章“使用 Node.js 进行 Web 开发”

这一章是本书的实践性章节，一步一步教你如何从零开始用 Express 框架创建一个网站，实现路由控制、模板解析、会话管理、数据库访问等功能，最终创建一个 Web 2.0 微博网站。

第 6 章“Node.js 进阶话题”

这一章涉及几个进阶话题，包括模块加载机制、控制流分析和优化、生产环境的应用部署等内容，最后还讨论了 Node.js 适用的范围，帮助读者在今后的开发中作出更好的取舍。

附录A“JavaScript 的高级特性”

这个附录介绍了 JavaScript 的一些高级特性，如函数作用域、闭包和对象的操作等内容。这些特性在浏览器端的JavaScript 开发中并没有受到应有的重视，而在 Node.js 中却十分常见，阅读这个附录可以帮助你更好地理解并运用 JavaScript 进行复杂的网站开发。

附录B“Node.js 编程规范”

这个附录介绍了 Node.js 代码风格的一些约定，遵守这些约定可以让你的代码更清晰、易懂，同时也有利于接口开发的统一。该附录还分享了一些开发经验，可以让程序避免很多意外错误和性能损失。

谁应该阅读本书

本书的目标读者是想要学习 Node.js，但没有任何系统的经验的开发者。如果你听说过 Node.js，并被它许多神奇的特性吸引，那么这本书就是为你准备的。通过阅读本书，你可以对 Node.js 有全面的认识，学会如何用 Node.js 编程，了解事件驱动、异步式 I/O 的编程模式，同时还可以掌握一些使用 JavaScript 进行函数式编程的方法。

本书假设读者已经学过至少一门编程语言，对基本的程序设计语言概念（如变量、函数、递归、对象）有所了解。如果你是首次学习编程语言，我建议你先学一门常见的且容易入门的语言，如 Java 或 C。

如何阅读本书

熟悉浏览器端 JavaScript 的读者将很容易学会 Node.js 的许多特性，包括事件式编程、闭包、回调函数等，因为这些特性已经在浏览器中被广泛应用。同时，你还可以学到 Node.js

在Web开发中的服务器端与浏览器端的结合方式，这无论是对前端设计还是后端开发都是有利的。你还会对 JavaScript 有一个全新的认识，因为服务端的 JavaScript 中没有 DOM 和 BOM，也不存在浏览器之间的兼容性问题。

不熟悉 JavaScript但是了解C、Java、C++、C#的读者将很容易学会 JavaScript 的语言特性及 Node.js 的基本机制，如模块和包。你需要关注的仅仅是 JavaScript 语言的特别之处，以及服务器端开发中需要注意的一些要点。

已经非常了解 Web 后端开发（如 PHP、ASP.net、Ruby on Rails、Django 等）的读者，本书将通过 Node.js 给你一个不同的视野。你会发现 Node.js 和这些传统的框架有很大的区别，因为它使用了事件式编程和异步 I/O，所以你需要改变一些已有的思维方式。同时，你还能享受到 Web 前后端紧密配合带来的新鲜感，并可能对 Ajax 有全新的认识。

如果是完全没有接触过JavaScript的读者，那么我建议你看完本书的前两章以后，花点时间到http://www.w3school.com.cn/js/网站看看 JavaScript 的入门教程。你只要了解基础知识就行了，本书并不要求你学成一个JavaScript专家。在这之后请阅读本书的附录A，了解一下实际开发中可能会遇到的稍微复杂的语言特性。附录A是为本书量身定做的，你可以从中很快地学会 Node.js 经常使用到的那些特性。如果你想更加深入系统地学习JavaScript，推荐阅读Mozilla JavaScript指南http://developer.mozilla.org/en/JavaScript/Guide。

本书从第3章开始，将介绍如何用 Node.js 开发，你应该仔细阅读这一章。第4章是一些最基本的模块介绍，涉及Node.js 模块的基本风格，这可能会帮助你理解后面介绍的 API。第5章是一个真枪实弹的实战演练，跟随这一章的每个步骤你就可以用 Node.js 实现一个真正的 Web 应用，体验开发的成就感。第6章则是一些进阶话题，你会在这里接触到 Node.js的一些深层次概念，同时你还将学会如何真正部署 Node.js 应用。

本书的每一章最后都有一个参考资料小节，里面有很多有价值的资料，如果感兴趣不妨继续深入阅读。在阅读本书的过程中，我建议你抽时间看看附录B，在这里你会了解到Node.js开发的一些编程规范，写出符合社区风格的漂亮程序。

如何学习 Node.js

通读本书，你将会学到 Node.js 的很多东西，但如果想完全掌握它，我建议你亲自尝试运行本书中的每一段代码。本书的所有代码可以在http://www.byvoid.com/project/node上找到。[1] 除此之外，你最好自己用 Node.js 做一个项目，因为通过实践你会遇到很多问题，解决这些问题可以大大加深对 Node.js 的理解。

注意，不要忘了互联网网上的资源，比如Node.js 的官方 API 文档http://nodejs.org/api/。我强烈推荐你去 CNodeJS 社区看看http://cnodejs.org/，这里汇集了许许多多中国优秀的

① 读者也可以到图灵社区（ituring.com.cn）本书的页面上下载源代码或提交勘误。——编者注

Node.js 开发者。他们每天都在讨论着大量有关Node.js各个方面的话题，你可以在上面获得很多帮助。同时，CNodeJS 社区的网站也是用 Node.js 写成的，而且是开源的，它是一个非常好的让你了解如何用 Node.js 开发网站的实例。

体例说明

本书正文中出现的代码引用都会以等宽字体标出，例如：`console.log('Node.js')`。代码段会以段落的形式用等宽字体显示，例如：

```
function hello() {
  console.log('Hello, world!');
}
```

在正文之中，偶尔还会穿插一些提示和警告，例如：

> **提示**　这是一个提示。

> **警告**　这是一个警告。

致谢

感谢对这本书提出宝贵意见的朋友们，他们是牟瞳、李垚[①]、周越、钟音、萧骐[②]、杨旭东、孙嘉龙、范泽一、宋文杰、续本达、田劲锋、孟亚兰和李宇亮。他们为本书的结构、内容、语言表述等方面给出了许多有建设性的意见。

感谢 CNodeJS 社区的贾超、田永强和微软亚洲研究院的杨懋，以及VMware公司的柴可夫。他们不仅帮助审阅了本书，还解决了许多技术问题，给这本书提出了许多改进方案。

感谢弓辰开发的 Rime 输入法[③]，我用它完成了本书的创作。

还要感谢我的朋友徐可可，图灵公司的杨海玲、谢工、王军花以及各位编辑，她们给我提供了许多帮助和鼓舞，没有她们的激励，我很难顶着巨大的学业压力坚持写完这本书。

郭家寶

---

① 李垚是果壳网的作者之一，他的个人网站是http://www.liyaos.com/。

② 萧骐是*Dive into Python*的译者，活跃在 linuxtoy http://linuxtoy.org/。

③ Rime 是一个优秀的开源输入法，它不仅支持繁体和简体的拼音输入，而且是跨平台的，可以在 Windows、Linux、Mac上使用，其网址是： http://code.google.com/p/rimeime/。

目　　录

第 1 章 Node.js简介

Node.js，或者 Node，是一个可以让 JavaScript 运行在服务器端的平台。它可以让 JavaScript 脱离浏览器的束缚运行在一般的服务器环境下，就像运行 Python、Perl、PHP、Ruby 程序一样。你可以用 Node.js 轻松地进行服务器端应用开发，Python、Perl、PHP、Ruby 能做的事情 Node.js 几乎都能做，而且可以做得更好。

Node.js 是一个为实时Web（Real-time Web）应用开发而诞生的平台，它从诞生之初就充分考虑了在实时响应、超大规模数据要求下架构的可扩展性。这使得它摒弃了传统平台依靠多线程来实现高并发的设计思路，而采用了单线程、异步式I/O、事件驱动式的程序设计模型。这些特性不仅带来了巨大的性能提升，还减少了多线程程序设计的复杂性，进而提高了开发效率。

Node.js 最初是由 Ryan Dahl 发起的开源项目，后来被 Joyent 公司注意到。Joyent 公司将 Ryan Dahl 招入旗下，因此现在的 Node.js 由 Joyent 公司管理并维护。尽管它诞生的时间（2009年）还不长，但它的周围已经形成了一个庞大的生态系统。Node.js 有着强大而灵活的包管理器（node package manager，npm），目前已经有上万个第三方模块，其中有网站开发框架，有 MySQL、PostgreSQL、MongoDB 数据库接口，有模板语言解析、CSS 生成工具、邮件、加密、图形、调试支持，甚至还有图形用户界面和操作系统 API工具。由 VMware 公司建立的云计算平台 Cloud Foundry 率先支持了 Node.js。2011年6月，微软宣布与 Joyent 公司合作，将 Node.js 移植到 Windows，同时 Windows Azure 云计算平台也支持 Node.js。Node.js 目前还处在迅速发展阶段，相信在不久的未来它一定会成为流行的Web应用开发平台。让我们从现在开始，一同探索 Node.js 的美妙世界吧！

1.1　Node.js 是什么

Node.js 不是一种独立的语言，与 PHP、Python、Perl、Ruby 的“既是语言也是平台”不同。Node.js 也不是一个 JavaScript 框架，不同于 CakePHP、Django、Rails。Node.js 更不是浏览器端的库，不能与 jQuery、ExtJS 相提并论。Node.js 是一个让 JavaScript 运行在服务端的开发平台，它让 JavaScript 成为脚本语言世界的一等公民，在服务端堪与 PHP、Python、Perl、Ruby 平起平坐。

Node.js 是一个划时代的技术，它在原有的 Web 前端和后端技术的基础上总结并提炼出了许多新的概念和方法，堪称是十多年来 Web 开发经验的集大成者。Node.js 可以作为服务器向用户提供服务，与 PHP、Python、Ruby on Rails 相比，它跳过了 Apache、Nginx 等 HTTP 服务器，直接面向前端开发。Node.js 的许多设计理念与经典架构（如 LAMP）有着很大的不同，可提供强大的伸缩能力，以适应21世纪10年代以后规模越来越庞大的互联网环境。

Node.js 与 JavaScript

说起 JavaScript，不得不让人想到浏览器。传统意义上，JavaScript 是由 ECMAScript、

文档对象模型（DOM）和浏览器对象模型（BOM）组成的，而 Mozilla 则指出 JavaScript 由 Core JavaScript 和 Client JavaScript 组成。之所以会有这种分歧，是因为 JavaScript 和浏览器之间复杂的历史渊源，以及其命途多舛的发展历程所共同造成的，我们会在后面详述。我们可以认为，Node.js 中所谓的 JavaScript 只是 Core JavaScript，或者说是 ECMAScript 的一个实现，不包含 DOM、BOM 或者 Client JavaScript。这是因为 Node.js 不运行在浏览器中，所以不需要使用浏览器中的许多特性。

Node.js 是一个让 JavaScript 运行在浏览器之外的平台。它实现了诸如文件系统、模块、包、操作系统 API、网络通信等 Core JavaScript 没有或者不完善的功能。历史上将 JavaScript 移植到浏览器外的计划不止一个，但Node.js 是最出色的一个。随着 Node.js 的成功，各种浏览器外的 JavaScript 实现逐步兴起，因此产生了 CommonJS 规范。CommonJS 试图拟定一套完整的 JavaScript 规范，以弥补普通应用程序所需的 API，譬如文件系统访问、命令行、模块管理、函数库集成等功能。CommonJS 制定者希望众多服务端 JavaScript 实现遵循 CommonJS 规范，以便相互兼容和代码复用。Node.js 的部份实现遵循了CommonJS规范，但由于两者还都处于诞生之初的快速变化期，也会有不一致的地方。

Node.js 的 JavaScript 引擎是 V8，来自 Google Chrome 项目。V8 号称是目前世界上最快的 JavaScript 引擎，经历了数次引擎革命，它的 JIT（Just-in-time Compilation，即时编译）执行速度已经快到了接近本地代码的执行速度。Node.js 不运行在浏览器中，所以也就不存在 JavaScript 的浏览器兼容性问题，你可以放心地使用 JavaScript 语言的所有特性。

1.2 Node.js 能做什么

正如 JavaScript 为客户端而生，Node.js 为网络而生。Node.js 能做的远不止开发一个网站那么简单，使用 Node.js，你可以轻松地开发：

- ❑ 具有复杂逻辑的网站；
- ❑ 基于社交网络的大规模 Web 应用；
- ❑ Web Socket 服务器；
- ❑ TCP/UDP 套接字应用程序；
- ❑ 命令行工具；
- ❑ 交互式终端程序；
- ❑ 带有图形用户界面的本地应用程序；
- ❑ 单元测试工具；
- ❑ 客户端 JavaScript 编译器。

Node.js 内建了 HTTP 服务器支持，也就是说你可以轻而易举地实现一个网站和服务器的组合。这和 PHP、Perl 不一样，因为在使用 PHP 的时候，必须先搭建一个 Apache 之类的

HTTP 服务器，然后通过 HTTP 服务器的模块加载或 CGI 调用，才能将 PHP 脚本的执行结果呈现给用户。而当你使用 Node.js 时，不用额外搭建一个 HTTP 服务器，因为 Node.js 本身就内建了一个。这个服务器不仅可以用来调试代码，而且它本身就可以部署到产品环境，它的性能足以满足要求。

Node.js 还可以部署到非网络应用的环境下，比如一个命令行工具。Node.js 还可以调用 C/C++ 的代码，这样可以充分利用已有的诸多函数库，也可以将对性能要求非常高的部分用 C/C++ 来实现。

1.3　异步式 I/O 与事件驱动

Node.js 最大的特点就是采用异步式 I/O 与事件驱动的架构设计。对于高并发的解决方案，传统的架构是多线程模型，也就是为每个业务逻辑提供一个系统线程，通过系统线程切换来弥补同步式 I/O 调用时的时间开销。Node.js 使用的是单线程模型，对于所有 I/O 都采用异步式的请求方式，避免了频繁的上下文切换。Node.js 在执行的过程中会维护一个事件队列，程序在执行时进入事件循环等待下一个事件到来，每个异步式 I/O 请求完成后会被推送到事件队列，等待程序进程进行处理。

例如，对于简单而常见的数据库查询操作，按照传统方式实现的代码如下：

```
res = db.query('SELECT * from some_table');
res.output();
```

以上代码在执行到第一行的时候，线程会阻塞，等待数据库返回查询结果，然后再继续处理。然而，由于数据库查询可能涉及磁盘读写和网络通信，其延时可能相当大（长达几个到几百毫秒，相比CPU的时钟差了好几个数量级），线程会在这里阻塞等待结果返回。对于高并发的访问，一方面线程长期阻塞等待，另一方面为了应付新请求而不断增加线程，因此会浪费大量系统资源，同时线程的增多也会占用大量的 CPU 时间来处理内存上下文切换，而且还容易遭受低速连接攻击。

看看Node.js是如何解决这个问题的：

```
db.query('SELECT * from some_table', function(res) {
  res.output();
});
```

这段代码中 `db.query` 的第二个参数是一个函数，我们称为*回调函数*。进程在执行到 `db.query` 的时候，不会等待结果返回，而是直接继续执行后面的语句，直到进入事件循环。当数据库查询结果返回时，会将事件发送到事件队列，等到线程进入事件循环以后，才会调用之前的回调函数继续执行后面的逻辑。

Node.js 的异步机制是基于事件的，所有的磁盘 I/O、网络通信、数据库查询都以非阻塞

的方式请求，返回的结果由事件循环来处理。图1-1 描述了这个机制。Node.js 进程在同一时刻只会处理一个事件，完成后立即进入事件循环检查并处理后面的事件。这样做的好处是，CPU 和内存在同一时间集中处理一件事，同时尽可能让耗时的 I/O 操作并行执行。对于低速连接攻击，Node.js 只是在事件队列中增加请求，等待操作系统的回应，因而不会有任何多线程开销，很大程度上可以提高 Web 应用的健壮性，防止恶意攻击。

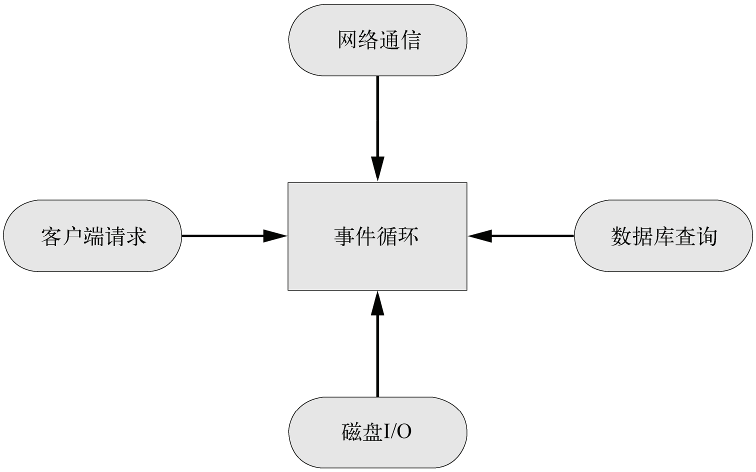


图1-1 事件循环

这种异步事件模式的弊端也是显而易见的，因为它不符合开发者的常规线性思路，往往需要把一个完整的逻辑拆分为一个个事件，增加了开发和调试难度。针对这个问题，Node.js 第三方模块提出了很多解决方案，我们会在第6章中详细讨论。

1.4 Node.js 的性能

1.4.1 Node.js 架构简介

Node.js 用异步式 I/O 和事件驱动代替多线程，带来了可观的性能提升。Node.js 除了使用 V8 作为JavaScript引擎以外，还使用了高效的 libev 和 libeio 库支持事件驱动和异步式 I/O。图1-2 是 Node.js 架构的示意图。

Node.js 的开发者在 libev 和 libeio 的基础上还抽象出了层 libuv。对于 POSIX[①]操作系统，libuv 通过封装 libev 和 libeio 来利用 epoll 或 kqueue。而在 Windows 下，libuv 使用了 Windows

① POSIX（Portable Operating System Interface）是一套操作系统 API 规范。一般而言，遵守 POSIX 规范的操作系统指的是 UNIX、Linux、Mac OS X 等。

的 IOCP（Input/Output Completion Port，输入输出完成端口）机制，以在不同平台下实现同样的高性能。

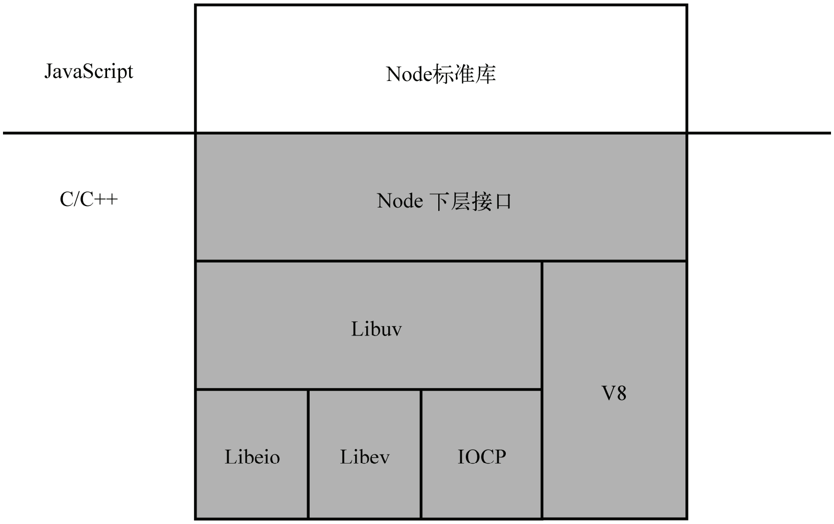


图1-2　Node.js 的架构

1.4.2　Node.js 与 PHP + Nginx

Snoopyxd 详细对比了 Node.js 与 PHP+Nginx 组合，结果显示在3000并发连接、30秒的测试下，输出"hello world"请求：

- PHP 每秒响应请求数为3624，平均每个请求响应时间为0.39秒；
- Node.js 每秒响应请求数为7677，平均每个请求响应时间为0.13秒。

而同样的测试，对MySQL查询操作：

- PHP 每秒响应请求数为1293，平均每个请求响应时间为0.82秒；
- Node.js 每秒响应请求数为2999，平均每个请求响应时间为0.33秒。

关于 Node.js 的性能优化及生产部署，我们会在第6章详细讨论。

1.5　JavaScript 简史

作为 Node.js 的基础，JavaScript 是一个完全为网络而诞生的语言。在今天看来，JavaScript 具有其他诸多语言不具备的优势，例如速度快、开销小、容易学习等，但在一开始它却并不是这样。多年以来，JavaScript 因为其低效和兼容性差而广受诟病，一直是一个被人嘲笑的"丑小鸭"，它在成熟之前经历了无数困难和坎坷，个中究竟，还要从它的诞生讲起。

1.5.1 Netscape 与 LiveScript

JavaScript 首次出现在1995年，正如现在的 Node.js 一样，当年 JavaScript 的诞生决不是偶然的。在1992年，一个叫 Nombas 的公司开发了“C减减”（C minus minus，Cmm）语言，后来改名为 ScriptEase。ScriptEase 最初的设计是将一种微型脚本语言与一个叫做 Espresso Page 的工具配合，使脚本能够在浏览器中运行，因此 ScriptEase 成为了第一个客户端脚本语言。

网景公司也想独立开发一种与 ScriptEase 相似的客户端脚本语言，Brendan Eich[①]接受了这一任务。起初这个语言的目标是为非专业的开发人员（如网站设计者），提供一个方便的工具。大多数网站设计者没有任何编程背景，因此这个语言应该尽可能简单、易学，最终一个弱类型的动态解释语言 LiveWire 就此诞生。LiveWire 没过多久就改名为 LiveScript 了，直到现在，在一些古老的 Web 页面中还能看到这个名字。

1.5.2 Java 与 Javascript

在JavaScript 诞生之前，Java applet[②]曾经被热炒。之前 Sun 公司一直在不遗余力地推广 Java，宣称 Java applet 将会改变人们浏览网页的方式。然而市场并没有像 Sun 公司预期的那样好，这很大程度上是因为 Java applet 速度慢而且操作不便。网景公司的市场部门抓住了这个机遇，与 Sun 合作完成了 LiveScript 实现，并在网景的Navigator 2.0 发布前，将 LiveScript 更名为 JavaScript。网景公司为了取得 Sun 公司的支持，把 JavaScript 称为 Java applet 和 HTML 的补充工具，目的之一就是为了帮助开发者更好地操纵 Java applet。

Netscape 决不会预料到当年那个市场策略带来的副作用有多大。多年来，到处都有人混淆 Java 和 JavaScript 这两个不相干的语言。两者除了名字相似和历史渊源之外，几乎没有任何关系。现在看来，从论坛到邮件列表，从网站到图书馆，能把 Java 和 JavaScript 区分开的倒是少数[③]。图1-3 是百度知道上的“Java 相关”分类。



图1-3 百度知道上的“Java 相关”分类

① Brendan Eich 被人称为 JavaScript 之父，他完全没想到自己当年无心设计的一个语言会成为今天最流行的网络脚本语言。

② applet 的意思是“小程序”，它是 Java 的一个客户端组件，需要在“容器”中运行，通常浏览器会充当这个容器。

③ Brendan Eich 为此抱憾不已，他后来在一个名为“ JavaScript at Ten Years”（JavaScript 这10年）的演讲稿中写道：“ Don’t let marketing name your language.”（不要为了营销决定语言名称）。

1.5.3　微软的加入—— JScript

就在网景公司如日中天之时，微软的 Internet Explorer 3 随 Windows 95 OSR2 捆绑销售的策略堪称一颗重磅炸弹，轻松击败了强劲的对手——网景公司的Navigator。尽管这个做法致使微软后来声名狼藉（以及一系列的反垄断诉讼），但 Internet Explorer3 的成功却有目共睹，其成功不仅仅在于市场营销策略，也源于产品本身。Internet Explorer 3 是一个划时代产品，因为它也实现了类似于 JavaScript 的客户端语言—— JScript，除此之外还有微软的“老本行”VBScript。JScript 的诞生成为 JavaScript 发展的一个重要里程碑，标志了动态网页时代的全面到来。图1-4 是 Windows 95 上的 Internet Explorer 3。

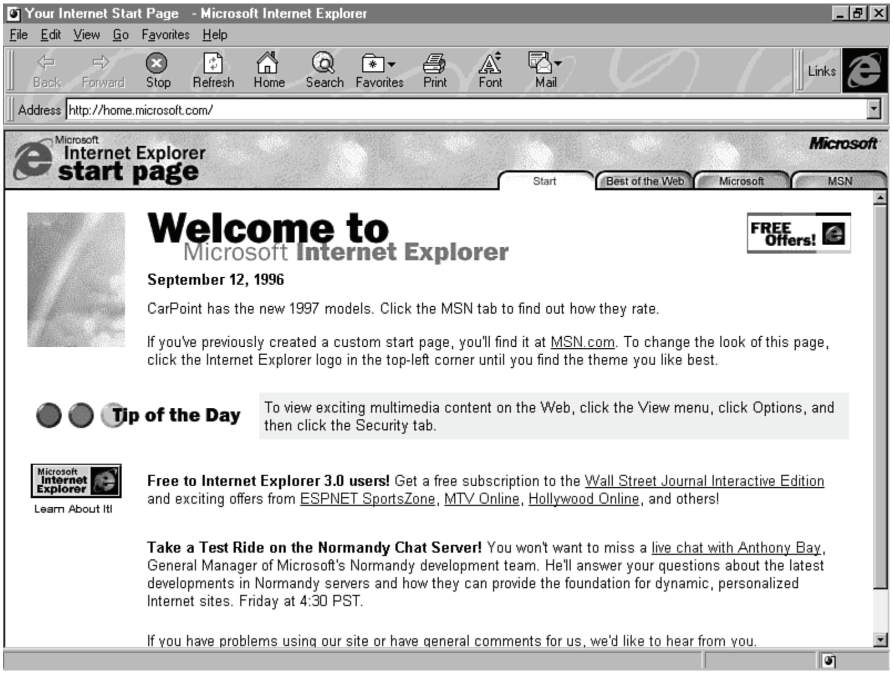


图1-4　Windows 95 上的 Internet Explorer 3

1.5.4　标准化—— ECMAScript

最初 JavaScript 并没有一个标准，因此在不同浏览器间有各种各样的兼容性的问题。Internet Explorer 占领市场以后这个问题变得更加尖锐，因此 JavaScript 的标准化势在必行。在1996年，JavaScript 标准由诸多软件厂商共同提交给ECMA（欧洲计算机制造商协会）。ECMA 通过了标准 ECMA-262，也就是 ECMAScript。紧接着国际标准化组织也采纳了 ECMAScript 标准（ISO-16262）。在接下来的几年里，浏览器开发者们就开始以 ECMAScript

作为规范来实现 JavaScript 解析引擎。

ECMAScript 诞生至今已经有了多个版本，最新的版本是在2009年12月发布的 ECMAScript 5，而到2012年为止，业界普遍支持的仍是 ECMAScript 3，只有新版的 Chrome 和 Firefox 实现了 ECMAScript 5。

提示 ECMAScript 仅仅是一个标准，而不是一个语言的具体实现，而且这个标准不像 C++ 语言规范那样严格而详细。除了 JavaScript 之外，ActionScript[①]、QtScript[②]、WMLScript[③]也是 ECMAScript 的实现。

1.5.5 浏览器兼容性问题

尽管有 ECMAScript 作为 JavaScript 的语法和语言特性标准，但是关于 JavaScript 其他方面的规范还是不明确，同时不同浏览器又加入了各自特有的对象、函数。这也就是为什么这么多年来同样的 JavaScript 代码会在不同的浏览器中呈现出不同的效果，甚至在一个浏览器中可以执行，而在另一个浏览器中却不可以。

要注意的是，浏览器的兼容性问题并不只是由 JavaScript 的兼容性造成的，而是 DOM、BOM、CSS 解析等不同的行为导致的。万维网联盟（World Wide Web Consortium，W3C）针对这个问题提出了很多标准建议，目前已经几乎被所有厂商和社区接受，浏览器的兼容性问题迅速得到了改善。

1.5.6 引擎效率革命和 JavaScript 的未来

第一款 JavaScript 引擎是由 Brendan Eich 在网景的 Navigator 中开发的，它的名字叫做 SpiderMonkey。SpiderMonkey 在这之后还用作 Mozilla Firefox 1.0~3.0版本的引擎，而从 Firefox 3.5 开始换为 TraceMonkey，4.0版本以后又换为 JaegerMonkey。Google Chrome 的 JavaScript 引擎是 V8，同时 V8 也是 Node.js 的引擎。微软从 Internet Explorer 9 开始使用其新的 JavaScript 引擎 Chakra。[④]

过去，JavaScript 一直不被人重视，很大程度上是因为它效率不高——不仅速度慢，还占用大量内存。但如今JavaScript的效率却令人刮目相看。历史总是如此相似，正如没有 Shockley 发明晶体管就没有电子科技革命一样，如果没有2008年以来的 JavaScript 引擎革命，Node.js 也不会这么快诞生。

---

① ActionScript 最初是 Adobe 公司 Flash 的一部分，用于控制动画效果，现在已经被广泛应用在 Adobe 的各项产品中。

② QtScript 是 Qt 4.3.0 以后引入的专用脚本工具。

③ WMLScript 是 WAP 协议的一部分，用于扩展 WML（Wireless Markup Language）页面。

④ 除此以外还有 KJS（用于 Konqueror）、Nitro（用于 Safari）、Carakan（用于Opera）等 JavaScript 引擎。

2008年 Mozilla Firefox 的一次改动，使 Firefox 3.0的 JavaScript 性能大幅提升，从而引发了 JavaScript 引擎之间的效率竞赛。紧接着 WebKit[①]开发团队宣告了 Safari 4 新的 JavaScript 引擎 SquirrelFish（后来改名 Nitro）可以大幅度提升脚本执行速度。Google Chrome 刚刚诞生就因它的 JavaScript 性能而备受称赞，但随着 WebKit 的 Squirrelfish Extreme 和 Mozilla 的 TraceMonkey 技术的出现，Chrome 的 JavaScript 引擎速度被超越了，于是 Chrome 2 发布时使用了更快速的 V8 引擎。V8 一出场就以其一骑绝尘般的速度打败了所有对手，一度成为 JavaScript 引擎的速度之王。于是其他浏览器的开发者开始奋力追赶，与以往不同的是，Internet Explorer 也加入了这次竞赛，并取得了不俗的成绩。

时至今日，各个 JavaScript 引擎的效率已经不相上下，通过不同引擎根据不同测试基准测得的结果各有千秋。更有趣的是，JavaScript 的效率在不知不觉中已经超越了其他所有传统的脚本语言，并带动了解释器的革新运动。JavaScript 已经成为了当今速度最快的脚本语言之一，昔日“丑小鸭”终于成了惊艳绝俗的“白天鹅”。

尽管如此，我们不能否认 JavaScript 还有很多不完美之处，譬如一些违反直觉的特性，这几乎成了 JavaScript 遭受批评和攻击的焦点。如今 JavaScript 还在继续发展，ECMAScript 6 也正在起草中，更有像 CoffeeScript 这样专门为了弥补 JavaScript 语言特性的不足而诞生的语言。Google 也专门针对客户端 JavaScript 不完美的地方推出了 Dart 语言。随着大规模的应用推广，我们有理由相信 JavaScript 会变得越来越好。

1.6　CommonJS

1.6.1　服务端 JavaScript 的重生

Node.js 并不是第一个尝试使 JavaScript 运行在浏览器之外的项目。追根溯源，在 JavaScript 诞生之初，网景公司就实现了服务端的 JavaScript，但由于需要支付一大笔授权费用才能使用，服务端 JavaScript 在当年并没有像客户端 JavaScript 一样流行开来。真正使大多数人见识到 JavaScript 在服务器开发威力的，是微软的 ASP。

2000年左右，也就是 ASP 蒸蒸日上的年代，很多开发者开始学习 JScript。然而 JScript 在当时并不是很受欢迎，一方面是早期的 JScript 和 JavaScript 兼容较差，另一方面微软大力推广的是 VBScript，而不是 JScript。随着后来 LAMP 的兴起，以及Web 2.0 时代的到来，Ajax 等一系列概念的提出，JavaScript 成了前端开发的代名词，同时服务端 JavaScript 也逐渐被人遗忘。

① WebKit 是苹果公司在设计 Safari 时开发的浏览器引擎，起源于 KHTML 和 KJS 项目的分支。WebKit 包含了一个网页引擎 WebCore 和一个脚本引擎 JavaScriptCore，但由于 JavaScript 引擎越来越独立，WebKit 逐渐成为了 WebCore 的代名词。

直至几年前，JavaScript 的种种优势才被重新提起，JavaScript 又具备了在服务端流行的条件，Node.js 应运而生。与此同时，RingoJS 也基于 Rhino 实现了类似的服务端 JavaScript 平台，还有像 CouchDB、MongoDB 等新型非关系型数据库也开始用 JavaScript 和 JSON 作为其数据操纵语言，基于 JavaScript 的服务端实现开始遍地开花。

1.6.2　CommonJS 规范与实现

正如当年为了统一 JavaScript 语言标准，人们制定了 ECMAScript 规范一样，如今为了统一 JavaScript 在浏览器之外的实现，CommonJS 诞生了。CommonJS 试图定义一套普通应用程序使用的API，从而填补 JavaScript 标准库过于简单的不足。CommonJS 的终极目标是制定一个像 C++ 标准库一样的规范，使得基于 CommonJS API 的应用程序可以在不同的环境下运行，就像用 C++ 编写的应用程序可以使用不同的编译器和运行时函数库一样。为了保持中立，CommonJS 不参与标准库实现，其实现交给像 Node.js 之类的项目来完成。图1-5 是 CommonJS 的各种实现。



图1-5　CommonJS 的实现

CommonJS 规范包括了模块（modules）、包（packages）、系统（system）、二进制（binary）、控制台（console）、编码（encodings）、文件系统（filesystems）、套接字（sockets）、单元测试（unit testing）等部分。目前大部分标准都在拟定和讨论之中，已经发布的标准有 Modules/1.0、Modules/1.1、Modules/1.1.1、Packages/1.0、System/1.0。

Node.js 是目前 CommonJS 规范最热门的一个实现，它基于 CommonJS 的 Modules/1.0 规范实现了 Node.js 的模块，同时随着 CommonJS 规范的更新，Node.js 也在不断跟进。由于目前 CommonJS 大部分规范还在起草阶段，Node.js 已经率先实现了一些功能，并将其反馈给 CommonJS 规范制定组织，但 Node.js 并不完全遵循 CommonJS 规范。这是所有规范制定者都会遇到的尴尬局面，因为规范的制定总是滞后于技术的发展。

1.7 参考资料

- Node.js: http://nodejs.org/。
- “再谈select、iocp、epoll、kqueue及各种I/O复用机制”: http://blog.csdn.net/shallwake/article/details/5265287。
- “巅峰对决：node.js和php性能测试”: http://snoopyxdy.blog.163.com/blog/static/60117440201183101319257/。
- “RingoJS vs. Node.js: Runtime Values”: http://hns.github.com/2010/09/21/benchmark. html。
- “Update on my Node.js Memory and GC Benchmark”: http://hns.github.com/2010/ 09/29/benchmark2.html。
- “JavaScript at Ten Years”: http://dl.acm.org/citation.cfm?id=1086382。
- QtScript： http://qt-project.org/doc/qt-4.8/qtscript.html。
- WebKit Open Source Project： http://www.webkit.org/。
- CommonJS API Specifications： http://www.commonjs.org/specs/。
- RingoJS： http://ringojs.org/。
- MongoDB：http://www.mongodb.org/。
- CouchDB：http://couchdb.apache.org/。
- Persevere： http://www.persvr.org/。
- 《JavaScript 语言精髓与编程实践》 周爱民著，电子工业出版社出版。
- 《JavaScript 高级程序设计（第3版）》 Nicholas C. Zakas 著，人民邮电出版社出版。
- 《JavaScript 权威指南（第5版）》 Flanagan David 著，机械工业出版社出版。

安装和配置Node.js

第 2 章

在使用 Node.js 开发之前，我们首先要配置好开发环境。本章的主要内容有：

- 如何在 Linux、Windows、Mac OS X 上通过包或包管理器安装 Node.js；
- 如何在 POSIX 和 Windows 下通过编译源代码安装 Node.js；
- 安装 npm（Node.js 包管理器）；
- 使用多版本管理器让多个 Node.js 的实例共存。

2.1　安装前的准备

Node.js 的生态系统建立在遵循 POSIX 标准的操作系统上，如 GNU/Linux、Mac OS X、Solaris 等。Node.js 起初不支持 Windows，只能运行在 cygwin 上，而0.6版本以后就支持 Windows 了，本节后面会详述。

从2009年诞生至今，Node.js 一直处在快速发展的时期，因此很多方法、技巧都会迅速被新的技术取代，本书内容也不例外。就在不久前，大家还都推荐通过编译源代码安装 Node.js，而现在已经有了成熟的安装包发行系统。我们推荐你尽量通过 Node.js 官方或操作系统发行版提供的途径进行安装，除非你想获得最新的版本，否则就不要费力编译了。

Windows 上的 Node.js

Node.js 从0.6版本开始可以运行在原生的 Windows 上了（不是 cygwin 或者其他虚拟环境）。这很大程度上应该归功于微软的合作，因为微软的云计算平台 Windows Azure 宣布了对 Node.js 完全支持。这对微软来说简直是破天荒的举动，因为一贯具有“开源死敌”之称的微软，竟然支持具有深厚开源血统的 Node.js，不得不令人瞠目结舌。

尽管如此，Node.js 与 Windows 的兼容性仍然不如 POSIX 操作系统，这一点在 npm 提供的第三方模块中体现得尤为突出。这主要是因为许多第三方的模块需要编译原生的 C/C++ 代码，其中编译框架和系统调用很多都是以 Linux 为范本的，与 Windows 不兼容。笔者不建议在 Windows 上进行 Node.js 开发或部署，当然出于学习目的，这些影响也是无关紧要的。相信随着 Node.js 的发展（以及微软与开源社区关系的进一步改善），Node.js 与 Windows 的兼容性会越来越好。

接下来的小节我们将详细介绍 Node.js 的安装方法。

2.2　快速安装

2.2.1　Microsoft Windows系统上安装Node.js

在 Windows 上安装 Node.js 十分方便，你只需要访问http://nodejs.org，点击Download链接，然后选择Windows Installer，下载安装包。下载完成后打开安装包（如图2-1所示），点击

Next即可自动完成安装。

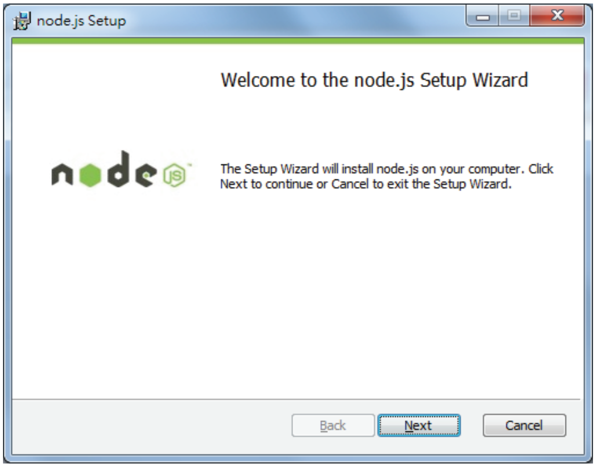


图2-1　在 Windows 上安装 Node.js

安装程序不会询问你安装路径，Node.js 会被自动安装到 C:\Program Files\nodejs 或 C:\Program Files (x86)\nodejs（64位系统）目录下，并且会在系统的 PATH 环境变量中增加该目录，因此我们可以在 Windows 的命令提示符中直接运行 `node`。

为了测试是否已经安装成功，我们在运行中输入 `cmd`，打开命令提示符，然后输入 `node`，将会进入 Node.js 的交互模式，如图2-2所示。

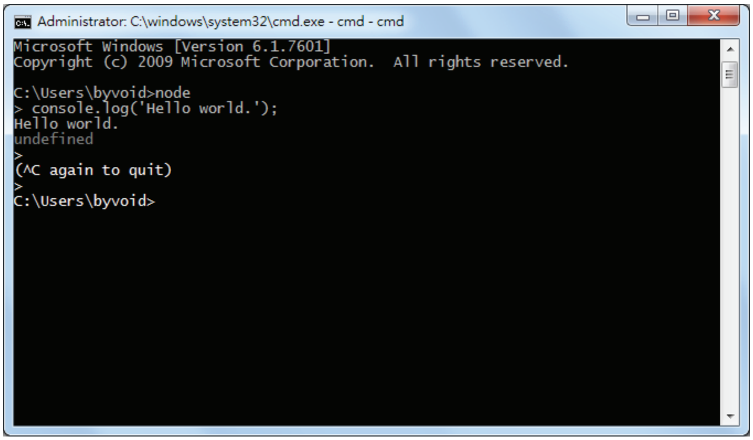


图2-2　Windows 命令提示符下的 Node.js

通过这种方式安装的 Node.js 还自动附带了 npm 图2-2，我们可以在命令提示符中直接输入 npm 来使用它。

2.2.2　Linux 发行版上安装Node.js

Node.js 目前还处在快速变化的时期，它的发行速度要远远大于 Linux 发行版维护的周期，因此各个 Linux 发行版官方的软件包管理器中提供的 Node.js 往往都比较过时。尽管如此，我们还是可以通过发行版的包管理器获得一个较为稳定的版本，根据不同的发行版，通过以下命令来获取Node.js，参见表2-1。

表2-1　在 Linux 发行版中获取 Node.js

Linux 发行版	命　令
Debian/Ubuntu	`apt-get install nodejs`
Fedora/RHEL/CentOS/Scientific Linux	`yum install nodejs`
openSUSE	`zypper install nodejs`
Arch Linux	`pacman -S nodejs`

如果你需要用软件包管理器来获得较新版本的 Node.js，就要根据不同的发行版选择第三方的软件源，具体请参阅：https://github.com/joyent/node/wiki/Installing-Node.js-via-package-manager。

2.2.3　Mac OS X上安装Node.js

Node.js 官方专门提供了 Mac OS X 的安装包，你可以在 http://nodejs.org 找到Download链接，然后选择Macintosh Installer，下载安装包。下载完成后运行安装包（如图2-3 所示），根据提示完成安装。

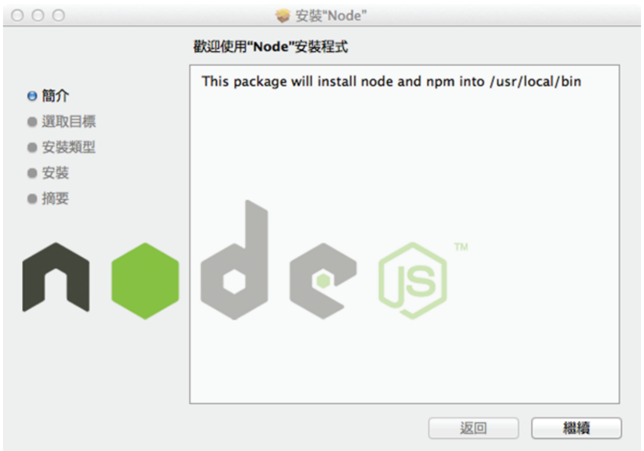


图2-3　在 Mac OS X 上安装 Node.js

Node.js 和 npm 会被安装到 /usr/local/bin 目录下，安装过程中需要系统管理员权限。安

装成功后你可以在终端机中运行 `node` 命令进入了 Node.js 的交互模式。如果出现 `-bash: node: command not found`，说明没有正确安装，需要重新运行安装包或者采取其他形式安装 Node.js。

2.3 编译源代码

Node.js 从 0.6 版本开始已经实现了源代码级别的跨平台，因此我们可以使用不同的编译命令将同一份源代码的基础上编译为不同平台下的原生可执行代码。

在编译之前，要先获取源码包。我们建议访问http://nodejs.org，点击Download链接，然后选择Source Code，下载正式发布的源码包。如果你需要开发中的版本，可以通过 https://github.com/joyent/node/zipball/master 获得，或者在命令行下输入`git clone git://github.com/joyent/node.git` 从git获得最新的分支。

2.3.1 在 POSIX 系统中编译

在 POSIX 系统中编译 Node.js 需要三个工具：

- C++编译器 gcc 或 clang/LLVM；
- Python 版本 2.5 以上，不支持 Python 3；
- libssl-dev 提供 SSL/TLS 加密支持。

如果你使用 Linux，那么你需要使用 `g++` 来编译 Node.js。在 Debian/Ubuntu 中，你可以通过 `apt-get install g++` 命令安装`g++`。在 Fedora/Redhat/CentOS 中，你可以使用 `yum install gcc-c++` 安装。

如果使用的是 Mac OS X，那么需要安装 Xcode。默认情况下，系统安装盘中会有 Xcode，可以从光盘中安装，或者访问 https://developer.apple.com/xcode/ 下载最新的版本。

Mac OS X 和几乎所有的 Linux 发行版都内置了 Python，你可以在终端机输入命令 `python --version` 检查 Python 的版本，可能会显示 `Python 2.7.2` 或其他版本。如果你发现版本号小于2.5或者直接出现了 `command not found`，那么你需要通过软件包管理器获得一个新版本的 Python，或者到 http://python.org/ 下载一个。

`libssl-dev` 是调用 OpenSSL 编译所需的头文件，用于提供 SSL/TLS 加密支持。Mac OS X 的 Xcode 内置了 `libssl-dev`。在 Debian/Ubuntu 中，你可以通过 `apt-get install libssl-dev` 命令安装。在 Fedora/Redhat/CentOS 中，你可以通过 `yum install openssl-devel` 命令安装。同样，你也可以访问 http://openssl.org/ 下载一个。

接下来，进入 Node.js 源代码所在目录，运行：

```
./configure
make
sudo make install
```

之后大约等待20分钟，Node.js 就安装完成了，而且附带安装了 npm。

如果你使用 Mac OS X，还可以尝试使用 homebrew 编译安装 Node.js。首先在 http://mxcl.github.com/homebrew/获取 homebrew，然后通过以下命令即可自动解析编译依赖并安装Node.js：

```
brew install node
```

2.3.2　在 Windows系统中编译

Node.js 在 Windows 下只能通过 Microsoft Visual Studio 编译，因此你需要首先安装 Visual Studio 或者免费的 Visual Studio Express。你还需要安装 Python 2（2.5以上的版本，但要小于 3.0），可以在http://python.org/取得。安装完 Python 以后请确保在PATH环境变量中添加 python.exe 所在的目录，如果没有则需要手动在"系统属性"中添加。

一切准备好以后，打开命令提示符，进入 Node.js 源代码所在的目录进行编译：

```
C:\Users\byvoid\node-v0.6.12>vcbuild.bat release
['-f', 'msvs', '-G', 'msvs_version=2010', '.\\node.gyp', '-I', '.\\common.gypi', '--depth=.',
  '-Dtarget_Project files generated.
C:\Program Files (x86)\MSBuild\Microsoft.Cpp\v4.0\Microsoft.CppBuild.targets(1151,5):
  warning MSB8012: http_parser.vcxproj -> C:\Users\byvoid\node-v0.6.12\
  Release\http_parser.lib
  js2c, and also js2c_experimental
  node_js2c
...
```

大约等待20分钟，编译完成。在 Release 子目录下面会有一个 node.exe 文件，这就是我们编译的**唯一目标**。也许有些令人惊讶，Node.js 编译后只有一个 node.exe文件，这说明 Node.js 的核心非常小巧精悍。直接运行 node.exe 即可进入 Node.js 的交互模式，在系统 PATH 环境变量中添加node.exe文件所在的目录，这样就可以在命令行中运行 `node` 命令了，剩下的工作就是手动安装 npm 了。

2.4　安装 Node 包管理器

Node 包管理器（npm）是一个由 Node.js 官方提供的第三方包管理工具，就像 PHP 的 Pear、Python 的 PyPI 一样。npm 是一个完全由 JavaScript 实现的命令行工具，通过 Node.js 执行，因此严格来讲它不属于 Node.js 的一部分。在最初的版本中，我们需要在安装完 Node.js 以后手动安装npm。但从 Node.js 0.6 开始，npm 已包含在发行包中了，我们在 Windows、Mac 上安装包和源代码包时会自动同时安装 npm。

如果你是在 Windows 下手动编译的，或是在 POSIX 系统中编译时指定了 `--without-npm` 参数，那就需要手动安装 npm 了。http://npmjs.org/提供了 npm 几种不同的安装方法，通常

你只需要执行以下命令：

```
curl http://npmjs.org/install.sh | sh
```

如果安装过程中出现了权限问题，那么需要在root权限下执行上面的语句，或者使用sudo。


```
curl http://npmjs.org/install.sh | sudo sh
```

其他安装方法，譬如从 git 中获取 npm 的最新分支，可以参考 http://npmjs.org/doc/README.html上的说明。

2.5 安装多版本管理器

迄今为止Node.js 更新速度还很快，有时候新版本还会将旧版本的一些 API 废除，以至于写好的代码不能向下兼容。有时候你可能想要尝试一下新版本有趣的特性，但又想要保持一个相对稳定的环境。基于这种需求，Node.js 的社区开发了多版本管理器，用于在一台机器上维护多个版本的 Node.js 实例，方便按需切换。Node 多版本管理器（Node Version Manager，nvm）是一个通用的叫法，它目前有许多不同的实现。通常我们说的 nvm 是指 https://github.com/creationix/nvm 或者 https://github.com/visionmedia/n。笔者根据个人偏好推荐使用 visionmedia/n，此小节就以它为例子介绍 Node 多版本管理器的用法。

n 是一个十分简洁的 Node 多版本管理器，就连它的名字也不例外。它的名字就是 n，没错，就一个字母。[①]

如果你已经安装好了 Node.js 和 npm 环境，就可以直接使用 npm install -g n 命令来安装 n。当然你可能会问：如果我想完全通过 n 来管理 Node.js，那么没安装之前哪来的 npm 呢？事实上，n 并不需要 Node.js 驱动，它只是 bash 脚本，使用 npm 安装只是采取一种简便的方式而已。我们可以在 https://github.com/visionmedia/n 下载它的代码，然后使用 make install 命令安装。

警告 n 不支持 Windows。

安装完 n 以后，在终端中运行 n --help 即可看到它的使用说明：

```
$ n --help

  Usage: n [options] [COMMAND] [config]
```

① 事实上，n 它曾经叫做 nvm，后来改名为 n。

```
Commands:

  n                            Output versions installed
  n latest [config ...]        Install or activate the latest node release
  n <version> [config ...]     Install and/or use node <version>
  n use <version> [args ...]   Execute node <version> with [args ...]
  n bin <version>              Output bin path for <version>
  n rm <version ...>           Remove the given version(s)
  n --latest                   Output the latest node version available
  n ls                         Output the versions of node available

Options:

  -V, --version   Output current version of n
  -h, --help      Display help information

Aliases:

  -       rm
  which   bin
  use     as
  list    ls
```

运行 n 版本号 可以安装任意已发布版本的 Node.js，n 会从http://nodejs.org下载源代码包，然后自动编译安装，例如：

```
$ n 0.7.5
######################################################################## 100.0%
{ 'target_defaults': { 'cflags': [],
                       'defines': [],
                       'include_dirs': [],
                       'libraries': ['-lz']},
  'variables': { 'host_arch': 'x64',
                 'node_install_npm': 'true',
                 'node_install_waf': 'true',
                 'node_prefix': '/usr/local/n/versions/0.7.5',
                 'node_shared_cares': 'false',
                 'node_shared_v8': 'false',
                 'node_use_dtrace': 'false',
                 'node_use_openssl': 'true',
                 'node_use_system_openssl': 'false',
                 'target_arch': 'x64',
                 'v8_use_snapshot': 'true'}}
creating  ./config.gypi
creating  ./config.mk
make -C out BUILDTYPE=Release
```

```
  CC(target) /usr/local/n/node-v0.7.5/out/Release/obj.target/http_parser/deps/
    http_parser/http_parser.o
  LIBTOOL-STATIC /usr/local/n/node-v0.7.5/out/Release/libhttp_parser.a
...
```


提示 通过 n 获取的 Node.js 实例都会安装在 /usr/local/n/versions/ 目录中。

之后再运行 n 即可列出已经安装的所有版本的 Node.js，其中“*”后的版本号为默认的 Node.js 版本，即可以直接使用 node 命令行调用的版本：

```
$ n
    0.6.11
  * 0.7.5
```

和安装新版本一样，运行 n 版本号 也可以在已安装的 Node.js 实例中切换环境，再运行 node 即为 n 指定的当前版本，例如：

```
$ n 0.6.11
  * 0.6.11
    0.7.5
$ node -v
v0.6.11
```

如果你不想切换默认环境，可以使用 n use 版本号 script.js 直接指定Node.js的运行实例，例如：

```
$ n use 0.6.11 script.js
```

警告 n 无法管理通过其他方式安装的 Node.js 版本实例（如官方提供的安装包、发行版软件源、手动编译），你必须通过 n 安装 Node.js 才能管理多版本的 Node.js。

关于 n 的更多细节，请访问它的项目主页 https://github.com/visionmedia/n获取信息。

2.6 参考资料

- “Building and Installing Node.js”: https://github.com/joyent/node/wiki/Installation。
- “Node package manager”: http://npmjs.org/doc/README.html。
- “Node version management”: https://github.com/visionmedia/n。

- “深入浅出Node.js（二）：Node.js & NPM的安装与配置”: http://www.infoq.com/cn/articles/nodejs-npm-install-config。
- “Node.js Now Runs Natively on Windows”: http://www.infoq.com/news/2011/11/Nodejs-Windows。
- 《Node Web开发》，David Herron著，人民邮电出版社出版。
- “如何在 Mac OS X Lion 上设定 node.js 的开发环境”: http://dreamerslab.com/blog/tw/how-to-setup-a-node- js-development-environment-on-mac-osx-lion/。

Node.js快速入门

第 3 章

Node.js 是一个方兴未艾的技术。一直以来，关于 Node.js 的宣传往往针对它“与众不同”的特性，这使得它显得格外扑朔迷离。事实上，Node.js的绝大部分特性跟大多数语言一样都是旧瓶装新酒，只是一些激进的特性使它显得很神秘。在这一章中，我们将会讲述Node.js的种种特性，让你对 Node.js 本身以及如何使用 Node.js 编程有一个全局性的了解，主要内容有：

- 编写第一个Node.js程序；
- 异步式I/O和事件循环；
- 模块和包；
- 调试。

让我们开始这个激动人心的旅程吧。

3.1 开始用 Node.js 编程

Node.js 具有深厚的开源血统，它诞生于托管了许多优秀开源项目的网站—— github。和大多数开源软件一样，它由一个黑客发起，然后吸引了一小拨爱好者参与贡献代码。一开始它默默无闻，靠口口相传扩散，直到某一天被一个黑客媒体曝光，进入业界视野，随后便有一些有远见的公司提供商业支持，使其逐步发展壮大。

用 Node.js 编程是一件令人愉快的事情，因为你将开始用黑客的思维和风格编写代码。你会发现像这样的语言是很容易入门的，可以快速了解到它的细节，然后掌握它。

3.1.1 Hello World

好了，让我们开始实现第一个 Node.js 程序吧。打开你常用的文本编辑器，在其中输入：

```
console.log('Hello World');
```

将文件保存为 helloworld.js，打开终端，进入 helloworld.js 所在的目录，执行以下命令：

```
node helloworld.js
```

如果一切正常，你将会在终端中看到输出 `Hello World`。很简单吧？下面让我们来解释一下这个程序的细节。`console` 是 Node.js 提供的控制台对象，其中包含了向标准输出写入的操作，如 `console.log`、`console.error` 等。`console.log` 是我们最常用的输出指令，它和 C 语言中的 `printf` 的功能类似，也可以接受任意多个参数，支持 `%d`、`%s` 变量引用，例如：

```
//consolelog.js

console.log('%s: %d', 'Hello', 25);
```

输出的是 Hello: 25。这只是一个简单的例子，如果你想了解 `console` 对象的详细功能，请参见 4.1.3节。

3.1.2 Node.js 命令行工具

在前面的 Hello World 示例中，我们用到了命令行中的 `node` 命令，输入 `node --help` 可以看到详细的帮助信息：

```
Usage: node [options] [ -e script | script.js ] [arguments]
       node debug script.js [arguments]

Options:
  -v, --version         print node's version
  -e, --eval script     evaluate script
  -p, --print           print result of --eval
  --v8-options          print v8 command line options
  --vars                print various compiled-in variables
  --max-stack-size=val set max v8 stack size (bytes)

Environment variables:
NODE_PATH               ';'-separated list of directories
                        prefixed to the module search path.
NODE_MODULE_CONTEXTS    Set to 1 to load modules in their own
                        global contexts.
NODE_DISABLE_COLORS     Set to 1 to disable colors in the REPL

Documentation can be found at http://nodejs.org/
```

其中显示了 `node` 的用法，运行 Node.js 程序的基本方法就是执行 `node` *`script.js`*，其中 *`script.js`*[①]是脚本的文件名。

除了直接运行脚本文件外，`node --help` 显示的使用方法中说明了另一种输出 Hello World 的方式：

```
$ node -e "console.log('Hello World');"
Hello World
```

我们可以把要执行的语句作为 `node -e` 的参数直接执行。

使用 `node` 的 REPL 模式

REPL（Read-eval-print loop），即输入—求值—输出循环。如果你用过 Python，就会知道在终端下运行无参数的 `python` 命令或者使用 Python IDLE 打开的 shell，可以进入一个即时求值的运行环境。Node.js 也有这样的功能，运行无参数的 `node` 将会启动一个 JavaScript 的交互式 shell：

---

① 事实上脚本文件的扩展名不一定是 .js，例如我们将脚本保存为 script.txt，使用 `node script.txt` 命令同样可以运行。扩展名使用 .js 只是一个约定而已，遵循了 JavaScript 脚本一贯的命名习惯。

```
$ node
> console.log('Hello World');
Hello World
undefined
> consol.log('Hello World');
ReferenceError: consol is not defined
    at repl:1:1
    at REPLServer.eval (repl.js:80:21)
    at repl.js:190:20
    at REPLServer.eval (repl.js:87:5)
    at Interface.<anonymous> (repl.js:182:12)
    at Interface.emit (events.js:67:17)
    at Interface._onLine (readline.js:162:10)
    at Interface._line (readline.js:426:8)
    at Interface._ttyWrite (readline.js:603:14)
    at ReadStream.<anonymous> (readline.js:82:12)
```

进入 REPL 模式以后，会出现一个“>”提示符提示你输入命令，输入后按回车，Node.js 将会解析并执行命令。如果你执行了一个函数，那么 REPL 还会在下面显示这个函数的返回值，上面例子中的 `undefined` 就是 `console.log` 的返回值。如果你输入了一个错误的指令，REPL 则会立即显示错误并输出调用栈。在任何时候，连续按两次 Ctrl + C 即可推出 Node.js 的 REPL 模式。

`node` 提出的 REPL 在应用开发时会给人带来很大的便利，例如我们可以测试一个包能否正常使用，单独调用应用的某一个模块，执行简单的计算等。

3.1.3　建立 HTTP 服务器

前面的 Hello World 程序对于你来说可能太简单了，因为这个例子几乎可以在任何语言的教科书上找到对应的内容，既无聊又乏味，让我们来点儿不一样的东西，真正感受一下 Node.js 的魅力所在吧。

Node.js 是为网络而诞生的平台，但又与 ASP、PHP 有很大的不同，究竟不同在哪里呢？如果你有 PHP 开发经验，会知道在成功运行 PHP 之前先要配置一个功能强大而复杂的 HTTP 服务器，譬如 Apache、IIS 或 Nginx，还需要将 PHP 配置为 HTTP 服务器的模块，或者使用 FastCGI 协议调用 PHP 解释器。这种架构是“浏览器 – HTTP 服务器 – PHP 解释器”的组织方式，而Node.js采用了一种不同的组织方式，如图3-1 所示。

我们看到，Node.js 将“HTTP服务器”这一层抽离，直接面向浏览器用户。这种架构从某种意义上来说是颠覆性的，因而会让人心存疑虑：Node.js作为HTTP服务器的效率足够吗？会不会提高耦合程度？我们不打算在这里讨论这种架构的利弊，后面章节会继续说明。

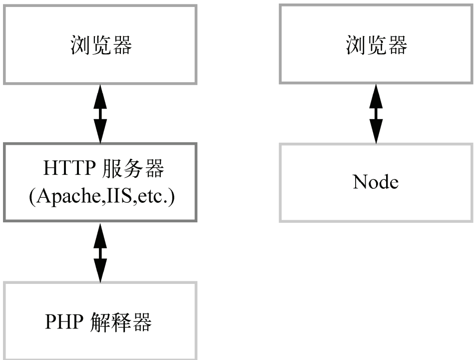


图3-1　Node.js 与 PHP 的架构

好了，回归正题，让我们创建一个 HTTP 服务器吧。建立一个名为 app.js 的文件，内容为：

```
//app.js

var http = require('http');

http.createServer(function(req, res) {
  res.writeHead(200, {'Content-Type': 'text/html'});
  res.write('<h1>Node.js</h1>');
  res.end('<p>Hello World</p>');
}).listen(3000);
console.log("HTTP server is listening at port 3000.");
```

接下来运行 `node app.js`命令，打开浏览器访问 http://127.0.0.1:3000，即可看到图3-2所示的内容。

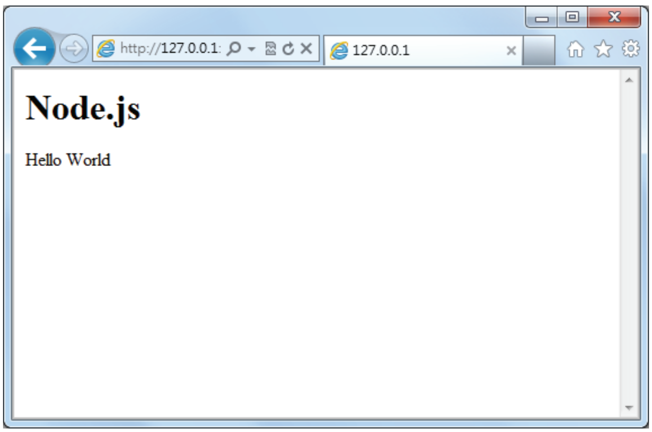


图3-2　用 Node.js 实现的 HTTP 服务器

用 Node.js 实现的最简单的 HTTP 服务器就这样诞生了。这个程序调用了 Node.js 提供的 `http` 模块，对所有 HTTP 请求答复同样的内容并监听 3000 端口。在终端中运行这个脚本时，我们会发现它并不像 Hello World 一样结束后立即退出，而是一直等待，直到按下 Ctrl + C 才会结束。这是因为 `listen` 函数中创建了事件监听器，使得 Node.js 进程不会退出事件循环。我们会在后面的章节中详细介绍这其中的奥秘。

小技巧——使用 supervisor

如果你有 PHP 开发经验，会习惯在修改 PHP 脚本后直接刷新浏览器以观察结果，而你在开发 Node.js 实现的 HTTP 应用时会发现，无论你修改了代码的哪一部份，都必须终止 Node.js 再重新运行才会奏效。这是因为 Node.js 只有在第一次引用到某部份时才会去解析脚本文件，以后都会直接访问内存，避免重复载入，而 PHP 则总是重新读取并解析脚本（如果没有专门的优化配置）。Node.js的这种设计虽然有利于提高性能，却不利于开发调试，因为我们在开发过程中总是希望修改后立即看到效果，而不是每次都要终止进程并重启。

supervisor 可以帮助你实现这个功能，它会监视你对代码的改动，并自动重启 Node.js。使用方法很简单，首先使用 npm 安装 supervisor：

```
$ npm install -g supervisor
```

如果你使用的是 Linux 或 Mac，直接键入上面的命令很可能会有权限错误。原因是 npm 需要把 supervisor 安装到系统目录，需要管理员授权，可以使用 `sudo npm install -g supervisor` 命令来安装。

接下来，使用 `supervisor` 命令启动 app.js：

```
$ supervisor app.js

DEBUG: Running node-supervisor with
DEBUG:   program 'app.js'
DEBUG:   --watch '.'
DEBUG:   --extensions 'node|js'
DEBUG:   --exec 'node'

DEBUG: Starting child process with 'node app.js'
DEBUG: Watching directory '/home/byvoid/.' for changes.
HTTP server is listening at port 3000.
```

当代码被改动时，运行的脚本会被终止，然后重新启动。在终端中显示的结果如下：

```
DEBUG: crashing child
DEBUG: Starting child process with 'node app.js'
HTTP server is listening at port 3000.
```

supervisor 这个小工具可以解决开发中的调试问题。

3.2 异步式 I/O 与事件式编程

Node.js 最大的特点就是异步式 I/O（或者非阻塞 I/O）与事件紧密结合的编程模式。这种模式与传统的同步式 I/O 线性的编程思路有很大的不同，因为控制流很大程度上要靠事件和回调函数来组织，一个逻辑要拆分为若干个单元。

3.2.1 阻塞与线程

什么是阻塞（block）呢？线程在执行中如果遇到磁盘读写或网络通信（统称为 I/O 操作），通常要耗费较长的时间，这时操作系统会剥夺这个线程的 CPU 控制权，使其暂停执行，同时将资源让给其他的工作线程，这种线程调度方式称为 *阻塞*。当 I/O 操作完毕时，操作系统将这个线程的阻塞状态解除，恢复其对CPU的控制权，令其继续执行。这种 I/O 模式就是通常的同步式 I/O（Synchronous I/O）或阻塞式 I/O （Blocking I/O）。

相应地，异步式 I/O （Asynchronous I/O）或非阻塞式 I/O （Non-blocking I/O）则针对所有 I/O 操作不采用阻塞的策略。当线程遇到 I/O 操作时，不会以阻塞的方式等待 I/O 操作的完成或数据的返回，而只是将 I/O 请求发送给操作系统，继续执行下一条语句。当操作系统完成 I/O 操作时，以事件的形式通知执行 I/O 操作的线程，线程会在特定时候处理这个事件。为了处理异步 I/O，线程必须有*事件循环*，不断地检查有没有未处理的事件，依次予以处理。

阻塞模式下，一个线程只能处理一项任务，要想提高吞吐量必须通过多线程。而非阻塞模式下，一个线程永远在执行计算操作，这个线程所使用的 CPU 核心利用率永远是 100%，I/O 以事件的方式通知。在阻塞模式下，多线程往往能提高系统吞吐量，因为一个线程阻塞时还有其他线程在工作，多线程可以让 CPU 资源不被阻塞中的线程浪费。而在非阻塞模式下，线程不会被 I/O 阻塞，永远在利用 CPU。多线程带来的好处仅仅是在多核 CPU 的情况下利用更多的核，而Node.js的单线程也能带来同样的好处。这就是为什么 Node.js 使用了单线程、非阻塞的事件编程模式。

图3-3 和图3-4 分别是多线程同步式 I/O 与单线程异步式 I/O 的示例。假设我们有一项工作，可以分为两个计算部分和一个 I/O 部分，I/O 部分占的时间比计算多得多（通常都是这样）。如果我们使用阻塞 I/O，那么要想获得高并发就必须开启多个线程。而使用异步式 I/O 时，单线程即可胜任。

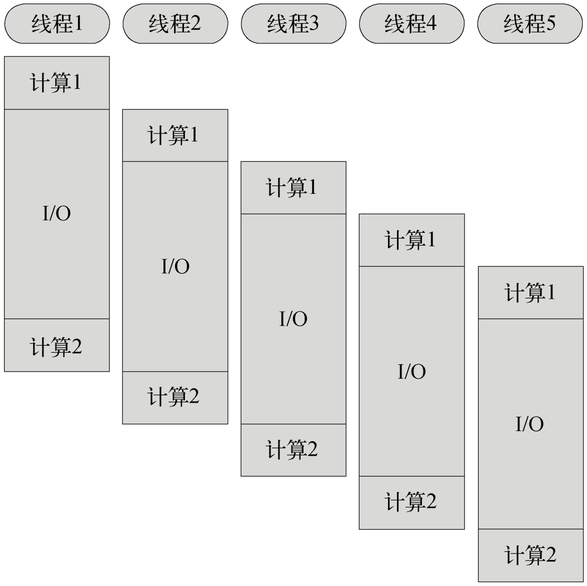


图3-3　多线程同步式 I/O

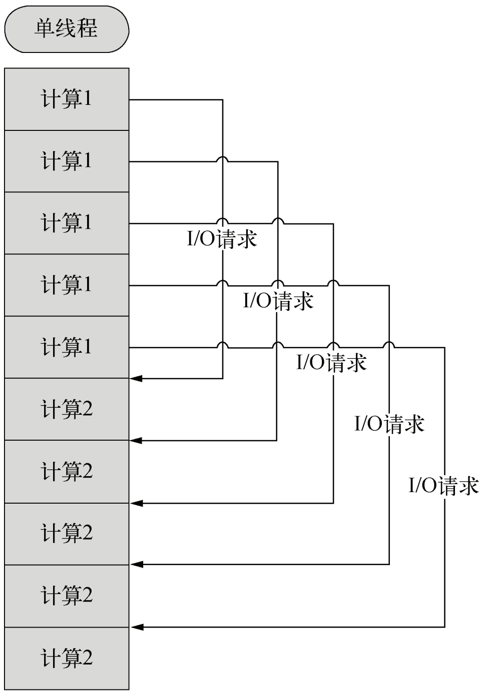


图3-4　单线程异步式 I/O

单线程事件驱动的异步式 I/O 比传统的多线程阻塞式 I/O 究竟好在哪里呢？简而言之，异步式 I/O 就是少了多线程的开销。对操作系统来说，创建一个线程的代价是十分昂贵的，需要给它分配内存、列入调度，同时在线程切换的时候还要执行内存换页，CPU 的缓存被清空，切换回来的时候还要重新从内存中读取信息，破坏了数据的局部性。①

当然，异步式编程的缺点在于不符合人们一般的程序设计思维，容易让控制流变得晦涩难懂，给编码和调试都带来不小的困难。习惯传统编程模式的开发者在刚刚接触到大规模的异步式应用时往往会无所适从，但慢慢习惯以后会好很多。尽管如此，异步式编程还是较为困难，不过可喜的是现在已经有了不少专门解决异步式编程问题的库（如async），参见6.2.2节。

表3-1比较了同步式 I/O 和异步式 I/O 的特点。

表3-1　同步式 I/O 和异步式 I/O 的特点

同步式 I/O（阻塞式）	异步式 I/O（非阻塞式）
利用多线程提供吞吐量	单线程即可实现高吞吐量
通过事件片分割和线程调度利用多核CPU	通过功能划分利用多核CPU
需要由操作系统调度多线程使用多核 CPU	可以将单进程绑定到单核 CPU
难以充分利用 CPU 资源	可以充分利用 CPU 资源
内存轨迹大，数据局部性弱	内存轨迹小，数据局部性强
符合线性的编程思维	不符合传统编程思维

3.2.2　回调函数

让我们看看在 Node.js 中如何用异步的方式读取一个文件，下面是一个例子：

```
//readfile.js

var fs = require('fs');
fs.readFile('file.txt', 'utf-8', function(err, data) {
  if (err) {
    console.error(err);
  } else {
    console.log(data);
  }
});
console.log('end.');
```

运行的结果如下：

```
end.
Contents of the file.
```

① 基于多线程的模型也有相应的解决方案，如轻量级线程（lightweight thread）等。事件驱动的单线程异步模型与多线程同步模型到底谁更好是一件非常有争议的事情，因为尽管消耗资源，后者的吞吐率并不比前者低。

Node.js 也提供了同步读取文件的 API：

```
//readfilesync.js

var fs = require('fs');
var data = fs.readFileSync('file.txt', 'utf-8');
console.log(data);
console.log('end.');
```

运行的结果与前面不同，如下所示：

```
$ node readfilesync.js
Contents of the file.
end.
```

同步式读取文件的方式比较容易理解，将文件名作为参数传入 `fs.readFileSync` 函数，阻塞等待读取完成后，将文件的内容作为函数的返回值赋给 `data` 变量，接下来控制台输出 `data` 的值，最后输出 `end.`。

异步式读取文件就稍微有些违反直觉了，`end.`先被输出。要想理解结果，我们必须先知道在 Node.js 中，异步式 I/O 是通过回调函数来实现的。`fs.readFile` 接收了三个参数，第一个是文件名，第二个是编码方式，第三个是一个函数，我们称这个函数为回调函数。JavaScript 支持匿名的函数定义方式，譬如我们例子中回调函数的定义就是嵌套在 `fs.readFile` 的参数表中的。这种定义方式在 JavaScript 程序中极为普遍，与下面这种定义方式实现的功能是一致的：

```
//readfilecallback.js

function readFileCallBack(err, data) {
  if (err) {
    console.error(err);
  } else {
    console.log(data);
  }
}

var fs = require('fs');
fs.readFile('file.txt', 'utf-8', readFileCallBack);
console.log('end.');
```

`fs.readFile` 调用时所做的工作只是将异步式 I/O 请求发送给了操作系统，然后立即返回并执行后面的语句，执行完以后进入事件循环监听事件。当 `fs` 接收到 I/O 请求完成的事件时，事件循环会主动调用回调函数以完成后续工作。因此我们会先看到 `end.`，再看到 file.txt 文件的内容。

警告 Node.js 中，并不是所有的 API 都提供了同步和异步版本。Node.js 不鼓励使用同步 I/O。

3.2.3 事件

Node.js 所有的异步 I/O 操作在完成时都会发送一个事件到事件队列。在开发者看来，事件由 `EventEmitter` 对象提供。前面提到的 `fs.readFile` 和 `http.createServer` 的回调函数都是通过 `EventEmitter` 来实现的。下面我们用一个简单的例子说明 `EventEmitter` 的用法：


```
//event.js

var EventEmitter = require('events').EventEmitter;
var event = new EventEmitter();

event.on('some_event', function() {
  console.log('some_event occured.');
});

setTimeout(function() {
  event.emit('some_event');
}, 1000);
```

运行这段代码，1秒后控制台输出了 `some_event occured.`。其原理是 `event` 对象注册了事件 `some_event` 的一个监听器，然后我们通过 `setTimeout` 在1000毫秒以后向 `event` 对象发送事件 `some_event`，此时会调用 `some_event` 的监听器。

我们将在 4.3.1节中详细讨论 `EventEmitter` 对象的用法。

Node.js 的事件循环机制

Node.js 在什么时候会进入事件循环呢？答案是 Node.js 程序由事件循环开始，到事件循环结束，所有的逻辑都是事件的回调函数，所以 Node.js 始终在事件循环中，程序入口就是事件循环第一个事件的回调函数。事件的回调函数在执行的过程中，可能会发出 I/O 请求或直接发射（emit）事件，执行完毕后再返回事件循环，事件循环会检查事件队列中有没有未处理的事件，直到程序结束。图3-5说明了事件循环的原理。

与其他语言不同的是，Node.js 没有显式的事件循环，类似 Ruby 的 `EventMachine::run()` 的函数在 Node.js 中是不存在的。Node.js 的事件循环对开发者不可见，由 `libev` 库实现。`libev` 支持多种类型的事件，如 `ev_io`、`ev_timer`、`ev_signal`、`ev_idle` 等，在 Node.js 中均被 `EventEmitter` 封装。`libev` 事件循环的每一次迭代，在 Node.js 中就是一次 Tick，`libev` 不断检查是否有活动的、可供检测的事件监听器，直到检测不到时才退出事件循环，进程结束。

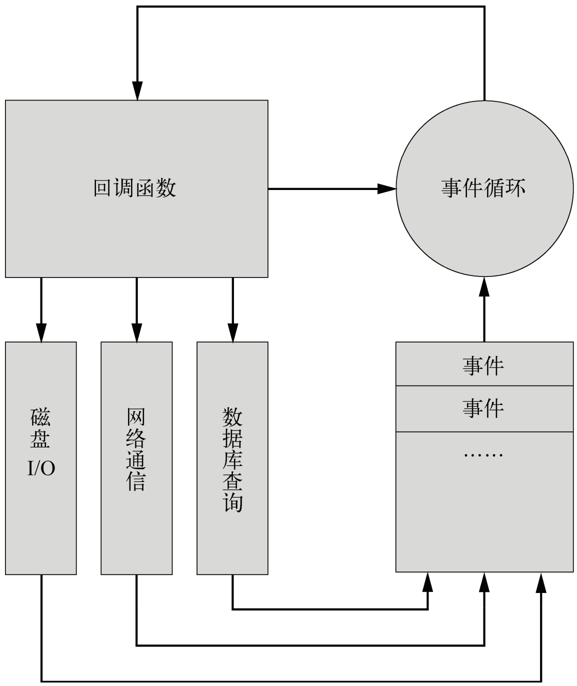


图3-5　事件循环

3.3　模块和包

模块（Module）和包（Package）是 Node.js 最重要的支柱。开发一个具有一定规模的程序不可能只用一个文件，通常需要把各个功能拆分、封装，然后组合起来，模块正是为了实现这种方式而诞生的。在浏览器 JavaScript 中，脚本模块的拆分和组合通常使用 HTML 的 `script` 标签来实现。Node.js 提供了 `require` 函数来调用其他模块，而且模块都是基于文件的，机制十分简单。

Node.js 的模块和包机制的实现参照了 CommonJS 的标准，但并未完全遵循。不过两者的区别并不大，一般来说你大可不必担心，只有当你试图制作一个除了支持 Node.js 之外还要支持其他平台的模块或包的时候才需要仔细研究。通常，两者没有直接冲突的地方。

我们经常把 Node.js 的模块和包相提并论，因为模块和包是没有本质区别的，两个概念也时常混用。如果要辨析，那么可以把包理解成是实现了某个功能模块的集合，用于发布和维护。对使用者来说，模块和包的区别是透明的，因此经常不作区分。本节中我们会详细介绍：

- 什么是模块；
- 如何创建并加载模块；
- 如何创建一个包；
- 如何使用包管理器；

3.3.1 什么是模块

模块是 Node.js 应用程序的基本组成部分，文件和模块是一一对应的。换言之，一个 Node.js 文件就是一个模块，这个文件可能是JavaScript 代码、JSON 或者编译过的 C/C++ 扩展。

在前面章节的例子中，我们曾经用到了 `var http = require('http')`，其中 `http` 是 Node.js 的一个核心模块，其内部是用 C++ 实现的，外部用 JavaScript 封装。我们通过 `require` 函数获取了这个模块，然后才能使用其中的对象。

3.3.2 创建及加载模块

介绍了什么是模块之后，下面我们来看看如何创建并加载它们。

1. 创建模块

在 Node.js 中，创建一个模块非常简单，因为一个文件就是一个模块，我们要关注的问题仅仅在于如何在其他文件中获取这个模块。Node.js 提供了 `exports` 和 `require` 两个对象，其中 `exports` 是模块公开的接口，`require` 用于从外部获取一个模块的接口，即所获取模块的 `exports` 对象。

让我们以一个例子来了解模块。创建一个 module.js 的文件，内容是：

```
//module.js

var name;

exports.setName = function(thyName) {
  name = thyName;
};

exports.sayHello = function() {
  console.log('Hello ' + name);
};
```

在同一目录下创建 getmodule.js，内容是：

```
//getmodule.js

var myModule = require('./module');
```

```
myModule.setName('BYVoid');
myModule.sayHello();
```

运行node getmodule.js，结果是：

```
Hello BYVoid
```

在以上示例中，module.js 通过 `exports` 对象把 `setName` 和 `sayHello` 作为模块的访问接口，在 getmodule.js 中通过 `require('./module')` 加载这个模块，然后就可以直接访问 module.js 中 `exports` 对象的成员函数了。

这种接口封装方式比许多语言要简洁得多，同时也不失优雅，未引入违反语义的特性，符合传统的编程逻辑。在这个基础上，我们可以构建大型的应用程序，npm 提供的上万个模块都是通过这种简单的方式搭建起来的。

2. 单次加载

上面这个例子有点类似于创建一个对象，但实际上和对象又有本质的区别，因为 `require` 不会重复加载模块，也就是说无论调用多少次 `require`，获得的模块都是同一个。我们在 getmodule.js 的基础上稍作修改：

```
//loadmodule.js

var hello1 = require('./module');
hello1.setName('BYVoid');

var hello2 = require('./module');
hello2.setName('BYVoid 2');

hello1.sayHello();
```

运行后发现输出结果是 `Hello BYVoid 2`，这是因为变量 `hello1` 和 `hello2` 指向的是同一个实例，因此 `hello1.setName` 的结果被 `hello2.setName` 覆盖，最终输出结果是由后者决定的。

3. 覆盖 exports

有时候我们只是想把一个对象封装到模块中，例如：

```
//singleobject.js

function Hello() {
  var name;

  this.setName = function (thyName) {
    name = thyName;
  };
```

```
  this.sayHello = function () {
    console.log('Hello ' + name);
  };
};

exports.Hello = Hello;
```

此时我们在其他文件中需要通过 require('./singleobject').Hello 来获取 Hello 对象，这略显冗余，可以用下面方法稍微简化：


```
//hello.js

function Hello() {
  var name;

  this.setName = function(thyName) {
    name = thyName;
  };

  this.sayHello = function() {
    console.log('Hello ' + name);
  };
};

module.exports = Hello;
```

这样就可以直接获得这个对象了：

```
//gethello.js

var Hello = require('./hello');

hello = new Hello();
hello.setName('BYVoid');
hello.sayHello();
```

注意，模块接口的唯一变化是使用 module.exports = Hello 代替了 exports.Hello= Hello。在外部引用该模块时，其接口对象就是要输出的 Hello 对象本身，而不是原先的 exports。

事实上，exports 本身仅仅是一个普通的空对象，即 {}，它专门用来声明接口，本质上是通过它为模块闭包[①]的内部建立了一个有限的访问接口。因为它没有任何特殊的地方，所以可以用其他东西来代替，譬如我们上面例子中的 Hello 对象。

① 闭包是函数式编程语言的常见特性，具体说明见本书附录A。

警告　不可以通过对 exports 直接赋值代替对 module.exports 赋值。exports 实际上只是一个和 module.exports 指向同一个对象的变量，它本身会在模块执行结束后释放，但 module 不会，因此只能通过指定 module.exports 来改变访问接口。

3.3.3　创建包

包是在模块基础上更深一步的抽象，Node.js 的包类似于 C/C++ 的函数库或者 Java/.Net 的类库。它将某个独立的功能封装起来，用于发布、更新、依赖管理和版本控制。Node.js 根据 CommonJS 规范实现了包机制，开发了 npm来解决包的发布和获取需求。

Node.js 的包是一个目录，其中包含一个 JSON 格式的包说明文件 package.json。严格符合 CommonJS 规范的包应该具备以下特征：

- package.json 必须在包的顶层目录下；
- 二进制文件应该在 bin 目录下；
- JavaScript 代码应该在 lib 目录下；
- 文档应该在 doc 目录下；
- 单元测试应该在 test 目录下。

Node.js 对包的要求并没有这么严格，只要顶层目录下有 package.json，并符合一些规范即可。当然为了提高兼容性，我们还是建议你在制作包的时候，严格遵守 CommonJS 规范。

1. 作为文件夹的模块

模块与文件是一一对应的。文件不仅可以是 JavaScript 代码或二进制代码，还可以是一个文件夹。最简单的包，就是一个作为文件夹的模块。下面我们来看一个例子，建立一个叫做 somepackage 的文件夹，在其中创建 index.js，内容如下：

```
//somepackage/index.js

exports.hello = function() {
  console.log('Hello.');
};
```

然后在 somepackage 之外建立 getpackage.js，内容如下：

```
//getpackage.js

var somePackage = require('./somepackage');

somePackage.hello();
```

运行 node getpackage.js，控制台将输出结果 `Hello.`。

我们使用这种方法可以把文件夹封装为一个模块，即所谓的包。包通常是一些模块的集合，在模块的基础上提供了更高层的抽象，相当于提供了一些固定接口的函数库。通过定制 package.json，我们可以创建更复杂、更完善、更符合规范的包用于发布。

2. package.json

在前面例子中的 somepackage 文件夹下，我们创建一个叫做 package.json 的文件，内容如下所示：

```
{
  "main" : "./lib/interface.js"
}
```

然后将 index.js 重命名为 interface.js 并放入 lib 子文件夹下。以同样的方式再次调用这个包，依然可以正常使用。

Node.js 在调用某个包时，会首先检查包中 package.json 文件的 `main` 字段，将其作为包的接口模块，如果 package.json 或 `main` 字段不存在，会尝试寻找 index.js 或 index.node 作为包的接口。

package.json 是 CommonJS 规定的用来描述包的文件，完全符合规范的 package.json 文件应该含有以下字段。

- `name`：包的名称，必须是唯一的，由小写英文字母、数字和下划线组成，不能包含空格。
- `description`：包的简要说明。
- `version`：符合语义化版本识别[①]规范的版本字符串。
- `keywords`：关键字数组，通常用于搜索。
- `maintainers`：维护者数组，每个元素要包含 `name`、`email`（可选）、`web`（可选）字段。
- `contributors`：贡献者数组，格式与`maintainers`相同。包的作者应该是贡献者数组的第一个元素。
- `bugs`：提交bug的地址，可以是网址或者电子邮件地址。
- `licenses`：许可证数组，每个元素要包含 `type`（许可证的名称）和 `url`（链接到许可证文本的地址）字段。
- `repositories`：仓库托管地址数组，每个元素要包含 `type`（仓库的类型，如 `git`）、`url`（仓库的地址）和 `path`（相对于仓库的路径，可选）字段。

① 语义化版本识别（Semantic Versioning）是由 Gravatars 和 GitHub 创始人 Tom Preston-Werner 提出的一套版本命名规范，最初目的是解决各式各样版本号大小比较的问题，目前被许多包管理系统所采用。

- `dependencies`：包的依赖，一个关联数组，由包名称和版本号组成。

下面是一个完全符合 CommonJS 规范的 package.json 示例：

```
{
  "name": "mypackage",
  "description": "Sample package for CommonJS. This package demonstrates the required
     elements of a CommonJS package.",
  "version": "0.7.0",
  "keywords": [
     "package",
     "example"
  ],
  "maintainers": [
     {
        "name": "Bill Smith",
        "email": "bills@example.com",
     }
  ],
  "contributors": [
     {
        "name": "BYVoid",
        "web": "http://www.byvoid.com/"
     }
  ],
  "bugs": {
     "mail": "dev@example.com",
     "web": "http://www.example.com/bugs"
  },
  "licenses": [
     {
        "type": "GPLv2",
        "url": "http://www.example.org/licenses/gpl.html"
     }
  ],
  "repositories": [
     {
        "type": "git",
        "url": "http://github.com/BYVoid/mypackage.git"
     }
  ],
  "dependencies": {
     "webkit": "1.2",
     "ssl": {
        "gnutls": ["1.0", "2.0"],
        "openssl": "0.9.8"
     }
  }
}
```

3.3.4 Node.js 包管理器

Node.js包管理器，即npm是 Node.js 官方提供的包管理工具[①]，它已经成了 Node.js 包的标准发布平台，用于 Node.js 包的发布、传播、依赖控制。npm 提供了命令行工具，使你可以方便地下载、安装、升级、删除包，也可以让你作为开发者发布并维护包。

1. 获取一个包

使用 npm 安装包的命令格式为：

```
npm [install/i] [package_name]
```

例如你要安装 `express`，可以在命令行运行：

```
$ npm install express
```

或者：

```
$ npm i express
```

随后你会看到以下安装信息：

```
npm http GET https://registry.npmjs.org/express
npm http 304 https://registry.npmjs.org/express
npm http GET https://registry.npmjs.org/mime/1.2.4
npm http GET https://registry.npmjs.org/mkdirp/0.3.0
npm http GET https://registry.npmjs.org/qs
npm http GET https://registry.npmjs.org/connect
npm http 200 https://registry.npmjs.org/mime/1.2.4
npm http 200 https://registry.npmjs.org/mkdirp/0.3.0
npm http 200 https://registry.npmjs.org/qs
npm http GET https://registry.npmjs.org/mime/-/mime-1.2.4.tgz
npm http GET https://registry.npmjs.org/mkdirp/-/mkdirp-0.3.0.tgz
npm http 200 https://registry.npmjs.org/mime/-/mime-1.2.4.tgz
npm http 200 https://registry.npmjs.org/mkdirp/-/mkdirp-0.3.0.tgz
npm http 200 https://registry.npmjs.org/connect
npm http GET https://registry.npmjs.org/formidable
npm http 200 https://registry.npmjs.org/formidable
express@2.5.8 ./node_modules/express
-- mime@1.2.4
-- mkdirp@0.3.0
-- qs@0.4.2
-- connect@1.8.5
```

此时 `express` 就安装成功了，并且放置在当前目录的 node_modules 子目录下。npm 在

① npm 之于 Node.js，就像 pip 之于 Python，gem 之于 Ruby，pear 之于 PHP，CPAN 之于 Perl……同时也像 apt-get 之于 Debian/Ubutnu，yum 之于 Fedora/RHEL/CentOS，homebrew 之于 Mac OS X。

获取 express 的时候还将自动解析其依赖，并获取 express 依赖的 mime、mkdirp、qs 和 connect。

2. 本地模式和全局模式

npm在默认情况下会从http://npmjs.org搜索或下载包，将包安装到当前目录的node_modules子目录下。

> **提示** 如果你熟悉 Ruby 的 gem 或者 Python 的 pip，你会发现 npm 与它们的行为不同，gem 或 pip 总是以全局模式安装，使包可以供所有的程序使用，而 npm 默认会把包安装到当前目录下。这反映了 npm 不同的设计哲学。如果把包安装到全局，可以提高程序的重复利用程度，避免同样的内容的多份副本，但坏处是难以处理不同的版本依赖。如果把包安装到当前目录，或者说本地，则不会有不同程序依赖不同版本的包的冲突问题，同时还减轻了包作者的 API 兼容性压力，但缺陷则是同一个包可能会被安装许多次。

在使用 npm 安装包的时候，有两种模式：本地模式和全局模式。默认情况下我们使用 npm install命令就是采用本地模式，即把包安装到当前目录的 node_modules 子目录下。Node.js 的 require 在加载模块时会尝试搜寻 node_modules 子目录，因此使用 npm 本地模式安装的包可以直接被引用。

npm 还有另一种不同的安装模式被成为全局模式，使用方法为：

```
npm [install/i] -g [package_name]
```

与本地模式的不同之处就在于多了一个参数 -g。我们在 介绍 supervisor那个小节中使用了 npm install -g supervisor 命令，就是以全局模式安装 supervisor。

为什么要使用全局模式呢？多数时候并不是因为许多程序都有可能用到它，为了减少多重副本而使用全局模式，而是因为本地模式不会注册 PATH 环境变量。举例说明，我们安装 supervisor 是为了在命令行中运行它，譬如直接运行 supervisor script.js，这时就需要在 PATH 环境变量中注册 supervisor。npm 本地模式仅仅是把包安装到 node_modules 子目录下，其中的 bin 目录没有包含在 PATH 环境变量中，不能直接在命令行中调用。而当我们使用全局模式安装时，npm 会将包安装到系统目录，譬如 /usr/local/lib/node_modules/，同时 package.json 文件中 bin 字段包含的文件会被链接到 /usr/local/bin/。/usr/local/bin/ 是在PATH 环境变量中默认定义的，因此就可以直接在命令行中运行 supervisor script.js命令了。

> **提示** 使用全局模式安装的包并不能直接在 JavaScript 文件中用 require 获得，因为 require 不会搜索 /usr/local/lib/node_modules/。我们会在第 6 章详细介绍模块的加载顺序。

本地模式和全局模式的特点如表3-2所示。

表3-2 本地模式与全局模式

模 式	可通过 require 使用	注册PATH
本地模式	是	否
全局模式	否	是

总而言之，当我们要把某个包作为工程运行时的一部分时，通过本地模式获取，如果要在命令行下使用，则使用全局模式安装。

提示 在 Linux/Mac 上使用 `npm install -g` 安装时有可能需要 root 权限，因为 /usr/local/lib/node_modules/ 通常只有管理员才有权写入。

3. 创建全局链接

npm 提供了一个有趣的命令 `npm link`，它的功能是在本地包和全局包之间创建符号链接。我们说过使用全局模式安装的包不能直接通过 `require` 使用，但通过 `npm link`命令可以打破这一限制。举个例子，我们已经通过 `npm install -g express` 安装了 `express`，这时在工程的目录下运行命令：

```
$ npm link express
./node_modules/express -> /usr/local/lib/node_modules/express
```

我们可以在 node_modules 子目录中发现一个指向安装到全局的包的符号链接。通过这种方法，我们就可以把全局包当本地包来使用了。

警告 `npm link` 命令不支持 Windows。

除了将全局的包链接到本地以外，使用 `npm link`命令还可以将本地的包链接到全局。使用方法是在包目录（ package.json 所在目录）中运行 `npm link` 命令。如果我们要开发一个包，利用这种方法可以非常方便地在不同的工程间进行测试。

4. 包的发布

npm 可以非常方便地发布一个包，比 pip、gem、pear 要简单得多。在发布之前，首先需要让我们的包符合 npm 的规范，npm 有一套以 CommonJS 为基础包规范，但与 CommonJS 并不完全一致，其主要差别在于必填字段的不同。通过使用 `npm init` 可以根据交互式问答产生一个符合标准的 package.json，例如创建一个名为 byvoidmodule 的目录，然后在这个目录中运行`npm init`：

```
$ npm init
Package name: (byvoidmodule) byvoidmodule
Description: A module for learning perpose.
Package version: (0.0.0) 0.0.1
Project homepage: (none) http://www.byvoid.com/
Project git repository: (none)
Author name: BYVoid
Author email: (none) byvoid.kcp@gmail.com
Author url: (none) http://www.byvoid.com/
Main module/entry point: (none)
Test command: (none)
What versions of node does it run on? (~0.6.10)
About to write to /home/byvoid/byvoidmodule/package.json

{
  "author": "BYVoid <byvoid.kcp@gmail.com> (http://www.byvoid.com/)",
  "name": "byvoidmodule",
  "description": "A module for learning perpose.",
  "version": "0.0.1",
  "homepage": "http://www.byvoid.com/",
  "repository": {
    "url": ""
  },
  "engines": {
    "node": "~0.6.12"
  },
  "dependencies": {},
  "devDependencies": {}
}

Is this ok? (yes) yes
```

这样就在 byvoidmodule 目录中生成一个符合 npm 规范的 package.json 文件。创建一个 index.js 作为包的接口，一个简单的包就制作完成了。

在发布前，我们还需要获得一个账号用于今后维护自己的包，使用 `npm adduser` 根据提示输入用户名、密码、邮箱，等待账号创建完成。完成后可以使用 `npm whoami` 测验是否已经取得了账号。

接下来，在 package.json 所在目录下运行 `npm publish`，稍等片刻就可以完成发布了。打开浏览器，访问 http://search.npmjs.org/ 就可以找到自己刚刚发布的包了。现在我们可以在世界的任意一台计算机上使用 `npm install byvoidmodule` 命令来安装它。图3-6 是npmjs.org上包的描述页面。

如果你的包将来有更新，只需要在 package.json 文件中修改 `version` 字段，然后重新使用 `npm publish` 命令就行了。如果你对已发布的包不满意（比如我们发布的这个毫无意义的包），可以使用 `npm unpublish` 命令来取消发布。

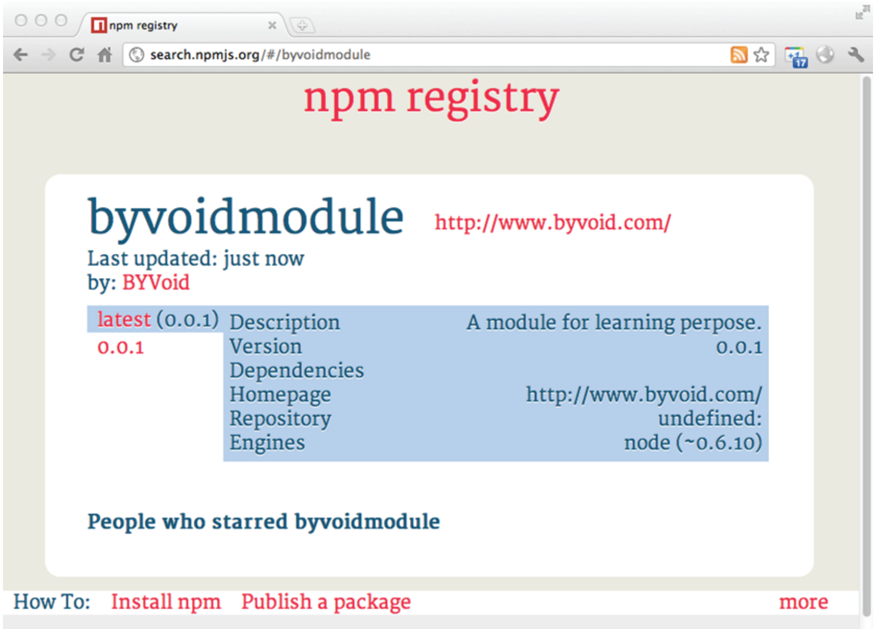


图3-6　在 npm 上发布的包

3.4　调试

写程序时免不了遇到 bug，而当 bug 发生以后，除了抓耳挠腮之外，一个常用的技术是单步调试。在写 C/C++ 程序的时候，我们有 Visual Studio、gdb 这样顺手的调试器，而脚本语言开发者就没有这么好的待遇了。多年以来，像 JavaScript 语言一直缺乏有效的调试手段，"攻城师"只能依靠"眼观六路，耳听八方"的方式进行静态查错，或者在代码之间添加冗长的输出语句来分析可能出错的地方。直到有了 FireBug、Chrome 开发者工具，JavaScript 才算有了基本的调试工具。在没有编译器或解译器的支持下，为缺乏内省机制的语言实现一个调试器是几乎不可能的。Node.js 的调试功能正是由 V8 提供的，保持了一贯的高效和方便的特性。尽管你也许已经对原始的调试方式十分适应，而且有了一套高效的调试技巧，但我们还是想介绍一下如何使用 Node.js 内置的工具和第三方模块来进行单步调试。

3.4.1　命令行调试

Node.js 支持命令行下的单步调试。下面是一个简单的程序：

```
var a = 1;
var b = 'world';
```

```
var c = function(x) {
  console.log('hello ' + x + a);
};
c(b);
```

在命令行下执行 node debug debug.js，将会启动调试工具：

```
< debugger listening on port 5858
connecting... ok
break in /home/byvoid/debug.js:1
  1 var a = 1;
  2 var b = 'world';
  3 var c = function(x) {
debug>
```

这样就打开了一个 Node.js 的调试终端，我们可以用一些基本的命令进行单步跟踪调试，参见表3-3。

表3-3　Node.js 调试命令

命　令	功　能
run	执行脚本，在第一行暂停
restart	重新执行脚本
cont, c	继续执行，直到遇到下一个断点
next, n	单步执行
step, s	单步执行并进入函数
out, o	从函数中步出
setBreakpoint(), sb()	在当前行设置断点
setBreakpoint('f()'), sb(...)	在函数f的第一行设置断点
setBreakpoint('script.js', 20), sb(...)	在 script.js 的第20行设置断点
clearBreakpoint, cb(...)	清除所有断点
backtrace, bt	显示当前的调用栈
list(5)	显示当前执行到的前后5行代码
watch(expr)	把表达式 expr 加入监视列表
unwatch(expr)	把表达式 expr 从监视列表移除
watchers	显示监视列表中所有的表达式和值
repl	在当前上下文打开即时求值环境
kill	终止当前执行的脚本
scripts	显示当前已加载的所有脚本
version	显示 V8 的版本

下面是一个简单的例子：

```
$ node debug debug.js
< debugger listening on port 5858
connecting... ok
break in /home/byvoid/debug.js:1
  1 var a = 1;
  2 var b = 'world';
  3 var c = function (x) {
debug> n
break in /home/byvoid/debug.js:2
  1 var a = 1;
  2 var b = 'world';
  3 var c = function (x) {
  4   console.log('hello ' + x + a);
debug> sb('debug.js', 4)
  1 var a = 1;
  2 var b = 'world';
  3 var c = function (x) {
* 4   console.log('hello ' + x + a);
  5 };
  6 c(b);
  7 });
debug> c
break in /home/byvoid/debug.js:4
  2 var b = 'world';
  3 var c = function (x) {
* 4   console.log('hello ' + x + a);
  5 };
  6 c(b);
debug> repl
Press Ctrl + C to leave debug repl
> x
'world'
> a + 1
2
debug> c
< hello world1
program terminated
```

3.4.2 远程调试

V8 提供的调试功能是基于 TCP 协议的，因此 Node.js 可以轻松地实现远程调试。在命令行下使用以下两个语句之一可以打开调试服务器：

```
node --debug[=port] script.js
node --debug-brk[=port] script.js
```

node --debug 命令选项可以启动调试服务器，默认情况下调试端口是 5858，也可以使用 --debug=1234 指定调试端口为 1234。使用 --debug 选项运行脚本时，脚本会正常执行，但不会暂停，在执行过程中调试客户端可以连接到调试服务器。如果要求脚本暂停执行等待客户端连接，则应该使用 --debug-brk 选项。这时调试服务器在启动后会立刻暂停执行脚本，等待调试客户端连接。

当调试服务器启动以后，可以用命令行调试工具作为调试客户端连接，例如：

```
//在一个终端中
$ node --debug-brk debug.js
debugger listening on port 5858

//在另一个终端中
$ node debug 127.0.0.1:5858
connecting... ok
debug> n
break in /home/byvoid/debug.js:2
  1 var a = 1;
  2 var b = 'world';
  3 var c = function (x) {
  4   console.log('hello ' + x + a);
debug>
```

事实上，当使用 node debug debug.js 命令调试时，只不过是用 Node.js 命令行工具将以上两步工作自动完成而已。

3.4.3　使用 Eclipse 调试 Node.js

基于 Node.js 的远程调试功能，我们甚至可以用支持 V8 调试协议的 IDE 调试，例如强大的 Eclipse。Eclipse 是深受广大"码农"喜爱的集成开发环境，有 Java 开发经验的对它一定不会陌生。在这一小节，我们将会学会如何使用 Eclipse 配置 Node.js 的调试环境，并实现单步调试功能。

1. 配置调试环境

在使用 Eclipse 之前，首先需要安装 JDK，可以在 http://www.oracle.com/technetwork/java/javase/downloads/index.html 获得，然后在 http://www.eclipse.org/downloads/ 取得一份 Eclipse。

启动 Eclipse，选择菜单栏中 Help→Install New Software...，此时会打开一个安装对话框，点击右边的按钮Add...，接下来会打开一个标题为Add Repository的对话框，在 Location 中输入 http://chromedevtools.googlecode.com/svn/update/dev/，Name 中输入 Chrome Developer，然后点击OK按钮。参见图3-7、图3-8和图3-9。

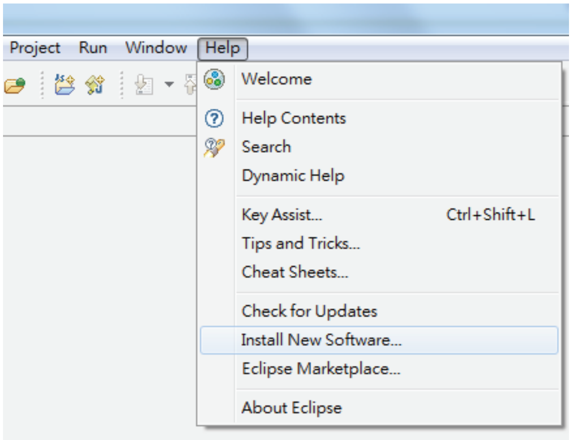


图3-7　Help→Install New Software...

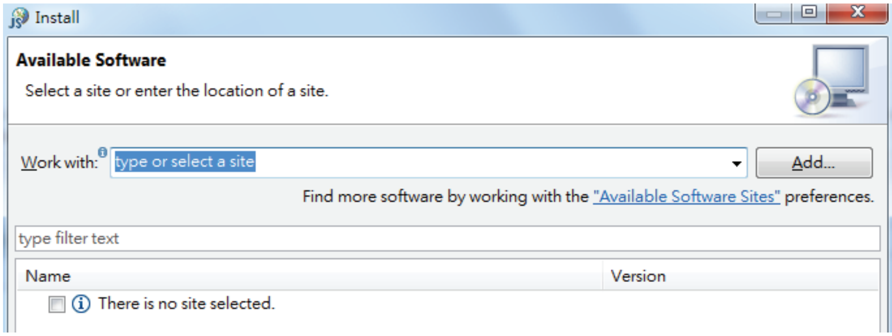


图3-8　Add...

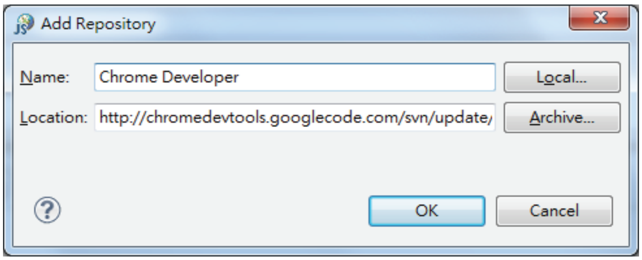


图3-9　Add Repository

然后在Work with后面的组合框中选择刚刚添加的Chrome Developer，等待片刻，在列表中选中Google Chrome Developer Tools，然后点击Next，参见图3-10。

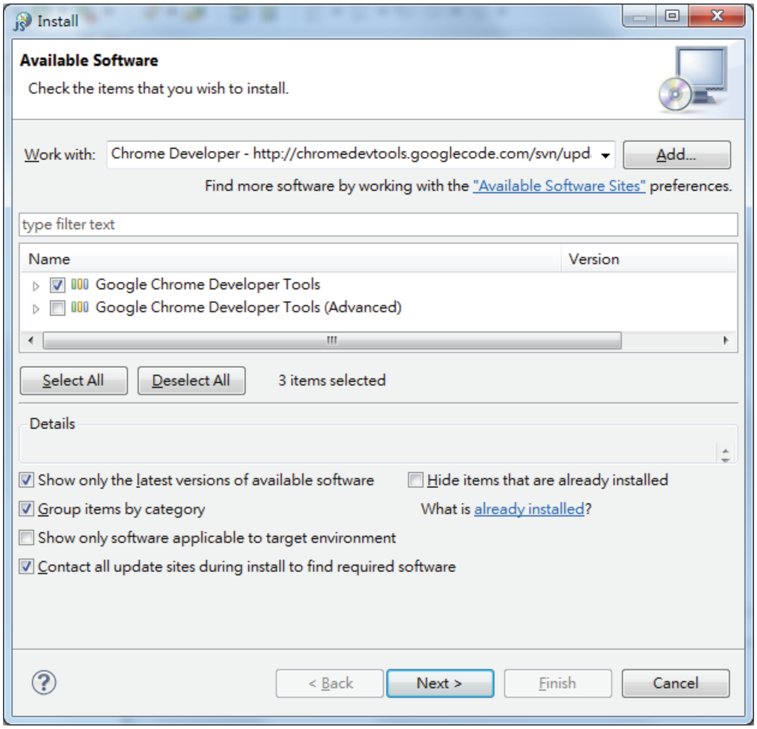


图3-10　Google Chrome Developer Tools

这时 Eclipse 会计算出所需安装的包和依赖，点击Next，参见图3-11。

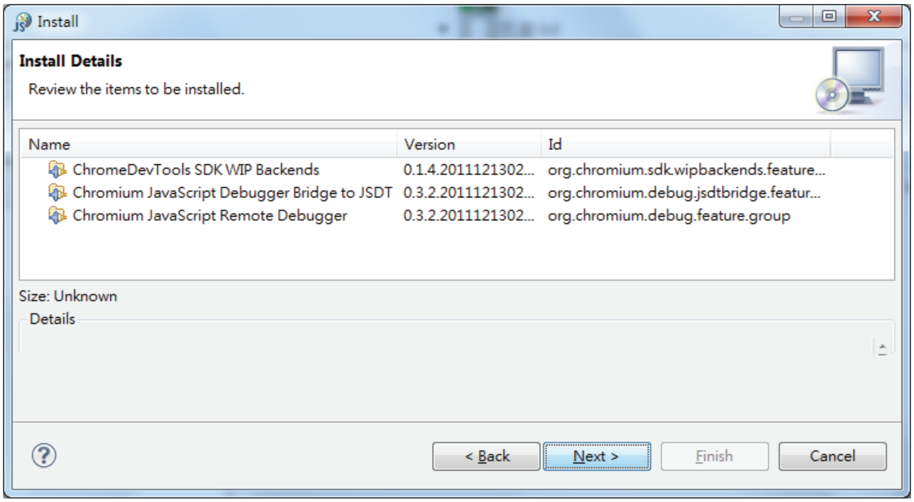


图3-11　计算依赖

阅读 License，选取I accept the terms of the license agreements，点击Next，参见图3-12。

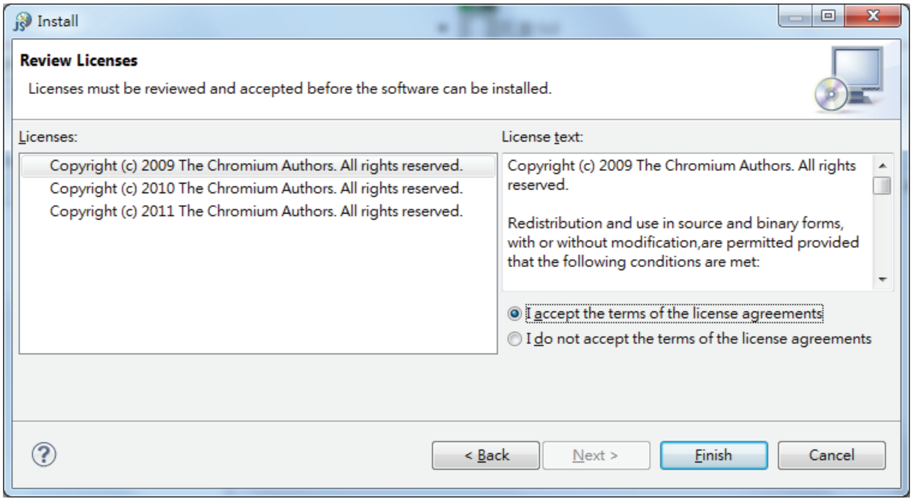


图3-12　License

接下来 Eclipse 会开始安装，稍等片刻，参见图3-13。

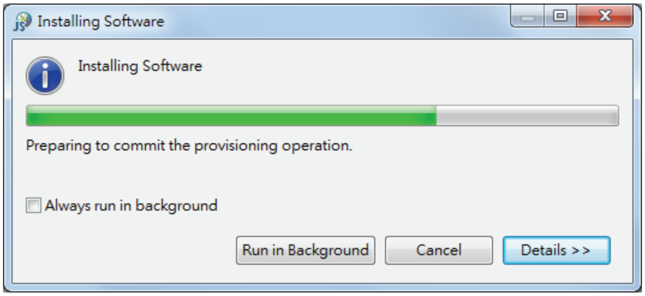


图3-13　安装过程

安装完成以后 Eclipse 会提示重新启动以应用更新，点击Restart Now，V8 调试工具就安装完成了，参见图3-14。

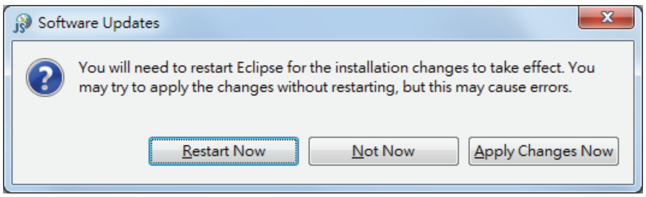


图3-14　Restart Now

2. 使用 Eclipse 调试 Node.js 程序

用 Eclipse 打开一个 Node.js 代码，选择Debug perspective进入调试视图，如图3-15所示。点击工具栏中 Debug 图标右边的向下三角形，选择Debug Configurations...（参见图3-16）。在配置窗口的左侧找到Standalone V8 VM，点击左上角的New图标，会产生一个新的配置。在配置中填写好Name，如NodeDebug，以及Host和Port。点击Apply应用配置，参见图3-17。

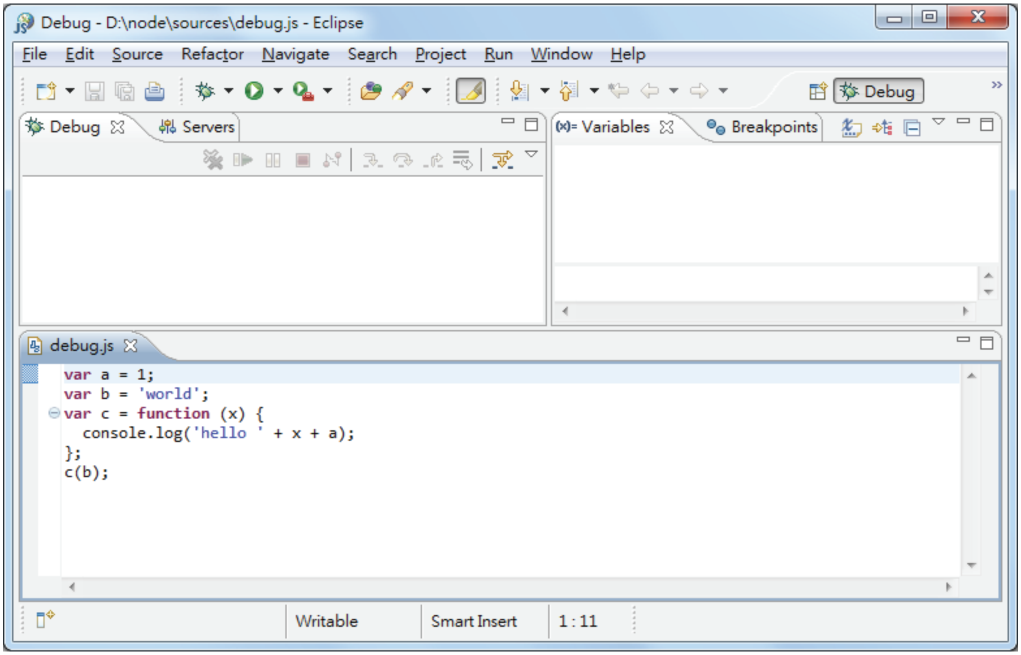


图3-15　Debug perspective

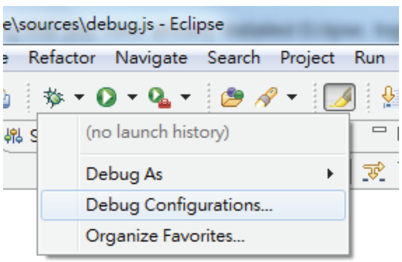


图3-16　Debug Configurations...

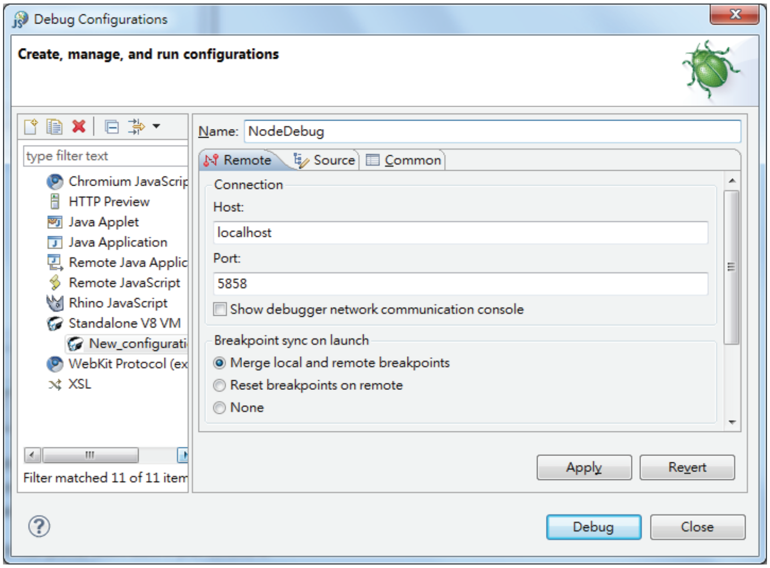


图3-17　配置 Standalone V8 VM

接下来，通过 `node --debug-brk=5858 debug.js` 命令启动要调试脚本的调试服务器，然后在 Eclipse 的工具栏中点击调试按钮，即可启动调试，如图3-18所示。

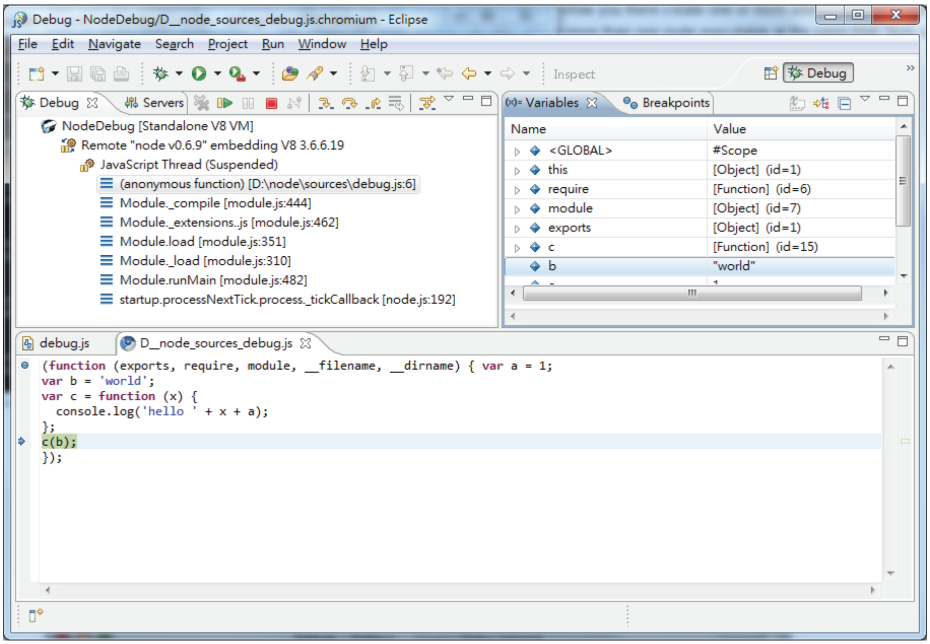


图3-18　启动调试

接下来你就可以随心所欲地使用 Eclipse 这个强大的 IDE 来调试 Node.js 脚本了。如果你对 Eclipse 比较熟悉，你会惊喜地发现 Eclipse 的所有单步调试、断点、监视功能均可以非常方便地使用。

3.4.4　使用 node-inspector 调试 Node.js

大部分基于 Node.js 的应用都是运行在浏览器中的，例如强大的调试工具 node-inspector。node-inspector 是一个完全基于 Node.js 的开源在线调试工具，提供了强大的调试功能和友好的用户界面，它的使用方法十分简便。

首先，使用 `npm install -g node-inspector` 命令安装 node-inspector，然后在终端中通过 `node --debug-brk=5858 debug.js` 命令连接你要除错的脚本的调试服务器，启动 node-inspector：

```
$ node-inspector
```

在浏览器中打开 http://127.0.0.1:8080/debug?port=5858，即可显示出优雅的 Web 调试工具，参见图3-19。

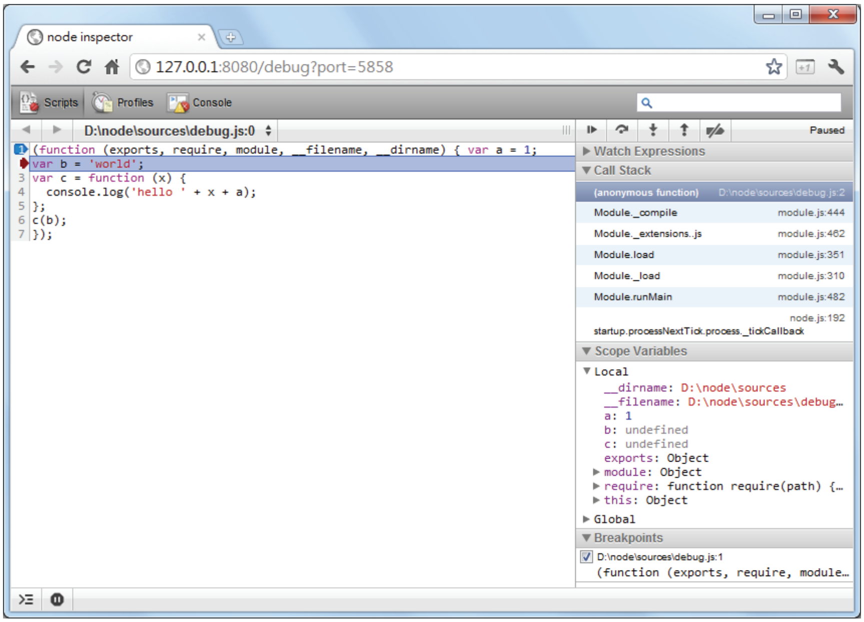


图3-19　node-inspector

node-inspector 的使用方法十分简单，和浏览器脚本调试工具一样，支持单步、断点、调用栈帧查看等功能。无论你以前有没有使用过调试工具，都可以在几分钟以内轻松掌握。

警告 node-inspector 使用了 WebKit Web Inspector，因此只能在 Chrome、Safari 等 WebKit 内核的浏览器中使用，而不支持 Firefox 或 Internet Explorer。

3.5 参考资料

- 《Node Web开发》，David Herron著，人民邮电出版社出版。
- node-supervisor: https://github.com/isaacs/node-supervisor。
- "Node.js is Cancer": http://teddziuba.com/2011/10/node-js-is-cancer.html。
- "Straight Talk on Event Loops": http://teddziuba.com/2011/10/straight-talk-on-event- loops.html。
- "nodejs 异步之 Timer & Tick 篇": http://club.cnodejs.org/topic/4f16442ccae1f4aa2700109b。
- "node.js成也异步，败也异步，评node.js的异步特性": http://www.jiangmiao.org/blog/2491.html。
- "被误解的 Node.js": https://www.ibm.com/developerworks/cn/web/1201_wangqf_nodejs/。libev: http://libev.schmorp.de。
- "深入浅出Node.js（三）：深入Node.js的模块机制": http://www.infoq.com/cn/articles/nodejs-module-mechanism。
- "npm中本地安装命令行类型的模块是不注册Path的": http://blog.goddyzhao.me/post/9835631010/no-direct-command-for-local-installed-command-line-modul。
- CommonJS 包/1.0: http://wiki.commonjs.org/wiki/Packages/1.0。
- Semantic Versioning 2.0.0-rc.1: http://semver.org/。
- "Symlink a package folder – npm": http://npmjs.org/doc/link.html。
- "Publish a package – npm": http://npmjs.org/doc/publish.html。
- "如何在Node.js中使用npm创建和发布一个模块": http://www.cnblogs.com/piyeyong/archive/2011/12/ 30/2308153.html。
- V8 debugger JSON based protocol: http://code.google.com/p/v8/wiki/DebuggerProtocol。
- node-inspector: https://github.com/dannycoates/node-inspector。

第 4 章 Node.js核心模块

核心模块是 Node.js 的心脏，它由一些精简而高效的库组成，为 Node.js 提供了基本的 API。本章中，我们挑选了一部分最常用的核心模块加以详细介绍，主要内容包括：

- 全局对象；
- 常用工具；
- 事件机制；
- 文件系统访问；
- HTTP 服务器与客户端。

4.1　全局对象

JavaScript 中有一个特殊的对象，称为*全局对象*（Global Object），它及其所有属性都可以在程序的任何地方访问，即*全局变量*。在浏览器 JavaScript 中，通常 `window` 是全局对象，而 Node.js 中的全局对象是 `global`，所有全局变量（除了 `global` 本身以外）都是 `global` 对象的属性。

我们在 Node.js 中能够直接访问到对象通常都是 `global` 的属性，如 `console`、`process` 等，下面逐一介绍。

4.1.1　全局对象与全局变量

`global` 最根本的作用是作为全局变量的宿主。按照 ECMAScript 的定义，满足以下条件的变量是全局变量：

- 在最外层定义的变量；
- 全局对象的属性；
- 隐式定义的变量（未定义直接赋值的变量）。

当你定义一个全局变量时，这个变量同时也会成为全局对象的属性，反之亦然。需要注意的是，在 Node.js 中你不可能在最外层定义变量，因为所有用户代码都是属于当前模块的，而模块本身不是最外层上下文。

提示　永远使用 `var` 定义变量以避免引入全局变量，因为全局变量会污染命名空间，提高代码的耦合风险。

4.1.2　process

`process` 是一个全局变量，即 `global` 对象的属性。它用于描述当前 Node.js 进程状态的对象，提供了一个与操作系统的简单接口。通常在你写本地命令行程序的时候，少不了要

和它打交道。下面将会介绍 process 对象的一些最常用的成员方法。

- process.argv是命令行参数数组，第一个元素是 node，第二个元素是脚本文件名，从第三个元素开始每个元素是一个运行参数。

```
console.log(process.argv);
```

将以上代码存储为 argv.js，通过以下命令运行：

```
$ node argv.js 1991 name=byvoid --v "Carbo Kuo"
[ 'node',
  '/home/byvoid/argv.js',
  '1991',
  'name=byvoid',
  '--v',
  'Carbo Kuo' ]
```


- process.stdout是标准输出流，通常我们使用的 console.log() 向标准输出打印字符，而 process.stdout.write() 函数提供了更底层的接口。
- process.stdin是标准输入流，初始时它是被暂停的，要想从标准输入读取数据，你必须恢复流，并手动编写流的事件响应函数。

```
process.stdin.resume();

process.stdin.on('data', function(data) {
  process.stdout.write('read from console: ' + data.toString());
});
```

- process.nextTick(callback)的功能是为事件循环设置一项任务，Node.js 会在下次事件循环调响应时调用 callback。

初学者很可能不理解这个函数的作用，有什么任务不能在当下执行完，需要交给下次事件循环响应来做呢？我们讨论过，Node.js 适合 I/O 密集型的应用，而不是计算密集型的应用，因为一个 Node.js 进程只有一个线程，因此在任何时刻都只有一个事件在执行。如果这个事件占用大量的 CPU 时间，执行事件循环中的下一个事件就需要等待很久，因此 Node.js 的一个编程原则就是尽量缩短每个事件的执行时间。process.nextTick() 提供了一个这样的工具，可以把复杂的工作拆散，变成一个个较小的事件。

```
function doSomething(args, callback) {
  somethingComplicated(args);
  callback();
}

doSomething(function onEnd() {
  compute();
});
```

我们假设 compute() 和 somethingComplicated() 是两个较为耗时的函数，以上的程序在调用 doSomething() 时会先执行 somethingComplicated()，然后立即调用回调函数，在 onEnd() 中又会执行 compute()。下面用 process.nextTick() 改写上面的程序：

```
function doSomething(args, callback) {
  somethingComplicated(args);
  process.nextTick(callback);
}

doSomething(function onEnd() {
  compute();
});
```

改写后的程序会把上面耗时的操作拆分为两个事件，减少每个事件的执行时间，提高事件响应速度。

警告　不要使用 setTimeout(fn,0)代替 process.nextTick(callback)，前者比后者效率要低得多。

我们探讨了process对象常用的几个成员，除此之外process还展示了process.platform、process.pid、process.execPath、process.memoryUsage() 等方法，以及 POSIX 进程信号响应机制。有兴趣的读者可以访问 http://nodejs.org/api/process.html 了解详细内容。

4.1.3　console

console 用于提供控制台标准输出，它是由 Internet Explorer 的 JScript 引擎提供的调试工具，后来逐渐成为浏览器的事实标准。Node.js 沿用了这个标准，提供与习惯行为一致的 console 对象，用于向标准输出流（stdout）或标准错误流（stderr）输出字符。

- console.log()：向标准输出流打印字符并以换行符结束。console.log 接受若干个参数，如果只有一个参数，则输出这个参数的字符串形式。如果有多个参数，则以类似于 C 语言 printf() 命令的格式输出。第一个参数是一个字符串，如果没有参数，只打印一个换行。

```
console.log('Hello world');
console.log('byvoid%diovyb');
console.log('byvoid%diovyb', 1991);
```

运行结果为：

```
Hello world
byvoid%diovyb
byvoid1991iovyb
```

- console.error()：与 console.log() 用法相同，只是向标准错误流输出。
- console.trace()：向标准错误流输出当前的调用栈。

```
console.trace();
```

运行结果为：

```
Trace:
    at Object.<anonymous> (/home/byvoid/consoletrace.js:1:71)
    at Module._compile (module.js:441:26)
    at Object..js (module.js:459:10)
    at Module.load (module.js:348:31)
    at Function._load (module.js:308:12)
    at Array.0 (module.js:479:10)
    at EventEmitter._tickCallback (node.js:192:40)
```


4.2 常用工具 util

util 是一个 Node.js 核心模块，提供常用函数的集合，用于弥补核心 JavaScript 的功能过于精简的不足。

4.2.1 util.inherits

util.inherits(constructor, superConstructor)是一个实现对象间原型继承的函数。JavaScript 的面向对象特性是基于原型的，与常见的基于类的不同。JavaScript 没有提供对象继承的语言级别特性，而是通过原型复制来实现的，具体细节我们在附录A中讨论，在这里我们只介绍 util.inherits 的用法，示例如下：

```
var util = require('util');

function Base() {
  this.name = 'base';
  this.base = 1991;

  this.sayHello = function() {
    console.log('Hello ' + this.name);
  };
}

Base.prototype.showName = function() {
  console.log(this.name);
```

```
};

function Sub() {
  this.name = 'sub';
}

util.inherits(Sub, Base);

var objBase = new Base();
objBase.showName();
objBase.sayHello();
console.log(objBase);

var objSub = new Sub();
objSub.showName();
//objSub.sayHello();
console.log(objSub);
```

我们定义了一个基础对象 Base 和一个继承自 Base 的 Sub，Base 有三个在构造函数内定义的属性和一个原型中定义的函数，通过 util.inherits 实现继承。运行结果如下：

```
base
Hello base
{ name: 'base', base: 1991, sayHello: [Function] }
sub
{ name: 'sub' }
```

注意，Sub 仅仅继承了 Base 在原型中定义的函数，而构造函数内部创造的 base 属性和 sayHello 函数都没有被 Sub 继承。同时，在原型中定义的属性不会被 console.log 作为对象的属性输出。如果我们去掉 objSub.sayHello(); 这行的注释，将会看到：

```
node.js:201
        throw e; // process.nextTick error, or 'error' event on first tick
              ^
TypeError: Object #<Sub> has no method 'sayHello'
    at Object.<anonymous> (/home/byvoid/utilinherits.js:29:8)
    at Module._compile (module.js:441:26)
    at Object..js (module.js:459:10)
    at Module.load (module.js:348:31)
    at Function._load (module.js:308:12)
    at Array.0 (module.js:479:10)
    at EventEmitter._tickCallback (node.js:192:40)
```

4.2.2　util.inspect

util.inspect(object,[showHidden],[depth],[colors])是一个将任意对象转换为字符串的方法，通常用于调试和错误输出。它至少接受一个参数 object，即要转换的对象。

showHidden 是一个可选参数，如果值为 true，将会输出更多隐藏信息。

depth 表示最大递归的层数，如果对象很复杂，你可以指定层数以控制输出信息的多少。如果不指定depth，默认会递归2层，指定为 null 表示将不限递归层数完整遍历对象。

如果color 值为 true，输出格式将会以 ANSI 颜色编码，通常用于在终端显示更漂亮的效果。

特别要指出的是，util.inspect 并不会简单地直接把对象转换为字符串，即使该对象定义了 toString 方法也不会调用。

```
var util = require('util');

function Person() {
  this.name = 'byvoid';

  this.toString = function() {
    return this.name;
  };
}

var obj = new Person();

console.log(util.inspect(obj));
console.log(util.inspect(obj, true));
```

运行结果是：

```
{ name: 'byvoid', toString: [Function] }
{ toString:
   { [Function]
     [prototype]: { [constructor]: [Circular] },
     [caller]: null,
     [length]: 0,
     [name]: '',
     [arguments]: null },
  name: 'byvoid' }
```

除了以上我们介绍的几个函数之外，util还提供了util.isArray()、util.isRegExp()、util.isDate()、util.isError() 四个类型测试工具，以及 util.format()、util.debug() 等工具。有兴趣的读者可以访问 http://nodejs.org/api/util.html 了解详细内容。

4.3 事件驱动 events

events 是 Node.js 最重要的模块，没有“之一”，原因是 Node.js 本身架构就是事件式的，而它提供了唯一的接口，所以堪称 Node.js 事件编程的基石。events 模块不仅用于用

户代码与 Node.js 下层事件循环的交互，还几乎被所有的模块依赖。

4.3.1　事件发射器

events 模块只提供了一个对象：events.EventEmitter。EventEmitter 的核心就是事件发射与事件监听器功能的封装。EventEmitter 的每个事件由一个事件名和若干个参数组成，事件名是一个字符串，通常表达一定的语义。对于每个事件，EventEmitter 支持若干个事件监听器。当事件发射时，注册到这个事件的事件监听器被依次调用，事件参数作为回调函数参数传递。

让我们以下面的例子解释这个过程：

```
var events = require('events');

var emitter = new events.EventEmitter();

emitter.on('someEvent', function(arg1, arg2) {
  console.log('listener1', arg1, arg2);
});

emitter.on('someEvent', function(arg1, arg2) {
  console.log('listener2', arg1, arg2);
});

emitter.emit('someEvent', 'byvoid', 1991);
```

运行的结果是：

```
listener1 byvoid 1991
listener2 byvoid 1991
```

以上例子中，emitter 为事件 someEvent 注册了两个事件监听器，然后发射了 someEvent 事件。运行结果中可以看到两个事件监听器回调函数被先后调用。

这就是EventEmitter最简单的用法。接下来我们介绍一下EventEmitter常用的API。

- EventEmitter.on(event, listener) 为指定事件注册一个监听器，接受一个字符串 event 和一个回调函数 listener。
- EventEmitter.emit(event, [arg1], [arg2], [...]) 发射 event 事件，传递若干可选参数到事件监听器的参数表。
- EventEmitter.once(event, listener) 为指定事件注册一个单次监听器，即监听器最多只会触发一次，触发后立刻解除该监听器。
- EventEmitter.removeListener(event, listener) 移除指定事件的某个监听器，listener 必须是该事件已经注册过的监听器。

- EventEmitter.removeAllListeners([event]) 移除所有事件的所有监听器，如果指定 event，则移除指定事件的所有监听器。

更详细的 API 文档参见 http://nodejs.org/api/events.html。

4.3.2 error 事件

EventEmitter 定义了一个特殊的事件 error，它包含了“错误”的语义，我们在遇到异常的时候通常会发射 error 事件。当 error 被发射时，EventEmitter 规定如果没有响应的监听器，Node.js 会把它当作异常，退出程序并打印调用栈。我们一般要为会发射 error 事件的对象设置监听器，避免遇到错误后整个程序崩溃。例如：

```
var events = require('events');

var emitter = new events.EventEmitter();

emitter.emit('error');
```

运行时会显示以下错误：

```
node.js:201
        throw e; // process.nextTick error, or 'error' event on first tick
              ^
Error: Uncaught, unspecified 'error' event.
    at EventEmitter.emit (events.js:50:15)
    at Object.<anonymous> (/home/byvoid/error.js:5:9)
    at Module._compile (module.js:441:26)
    at Object..js (module.js:459:10)
    at Module.load (module.js:348:31)
    at Function._load (module.js:308:12)
    at Array.0 (module.js:479:10)
    at EventEmitter._tickCallback (node.js:192:40)
```

4.3.3 继承 EventEmitter

大多数时候我们不会直接使用 EventEmitter，而是在对象中继承它。包括 fs、net、http 在内的，只要是支持事件响应的核心模块都是 EventEmitter 的子类。

为什么要这样做呢？原因有两点。首先，具有某个实体功能的对象实现事件符合语义，事件的监听和发射应该是一个对象的方法。其次 JavaScript 的对象机制是基于原型的，支持部分多重继承，继承 EventEmitter 不会打乱对象原有的继承关系。

4.4 文件系统 fs

fs 模块是文件操作的封装，它提供了文件的读取、写入、更名、删除、遍历目录、链

4

接等 POSIX 文件系统操作。与其他模块不同的是，fs 模块中所有的操作都提供了异步的和同步的两个版本，例如读取文件内容的函数有异步的 fs.readFile() 和同步的 fs.readFileSync()。我们以几个函数为代表，介绍 fs 常用的功能，并列出 fs 所有函数的定义和功能。

4.4.1 fs.readFile

fs.readFile(filename,[encoding],[callback(err,data)])是最简单的读取文件的函数。它接受一个必选参数 filename，表示要读取的文件名。第二个参数 encoding 是可选的，表示文件的字符编码。callback 是回调函数，用于接收文件的内容。如果不指定 encoding，则 callback 就是第二个参数。回调函数提供两个参数 err 和 data，err 表示有没有错误发生，data 是文件内容。如果指定了 encoding，data 是一个解析后的字符串，否则 data 将会是以 Buffer 形式表示的二进制数据。

例如以下程序，我们从 content.txt 中读取数据，但不指定编码：

```
var fs = require('fs');

fs.readFile('content.txt', function(err, data) {
  if (err) {
    console.error(err);
  } else {
    console.log(data);
  }
});
```

假设 content.txt 中的内容是 UTF-8 编码的 Text 文本文件示例，运行结果如下：

```
<Buffer 54 65 78 74 20 e6 96 87 e6 9c ac e6 96 87 e4 bb b6 e7 a4 ba e4 be 8b>
```

这个程序以二进制的模式读取了文件的内容，data 的值是 Buffer 对象。如果我们给 fs.readFile 的 encoding 指定编码：

```
var fs = require('fs');

fs.readFile('content.txt', 'utf-8', function(err, data) {
  if (err) {
    console.error(err);
  } else {
    console.log(data);
  }
});
```

那么运行结果则是：

```
Text 文本文件示例
```

当读取文件出现错误时，err 将会是 Error 对象。如果 content.txt 不存在，运行前面的代码则会出现以下结果：

```
{ [Error: ENOENT, no such file or directory 'content.txt'] errno: 34, code: 'ENOENT',
path: 'content.txt' }
```

提示 Node.js 的异步编程接口习惯是以函数的最后一个参数为回调函数，通常一个函数只有一个回调函数。回调函数是实际参数中第一个是 err，其余的参数是其他返回的内容。如果没有发生错误，err 的值会是 null 或 undefined。如果有错误发生，err 通常是 Error 对象的实例。

4.4.2 fs.readFileSync

fs.readFileSync(filename, [encoding])是 fs.readFile 同步的版本。它接受的参数和 fs.readFile 相同，而读取到的文件内容会以函数返回值的形式返回。如果有错误发生，fs 将会抛出异常，你需要使用 try 和 catch 捕捉并处理异常。

提示 与同步 I/O 函数不同，Node.js 中异步函数大多没有返回值。

4.4.3 fs.open

fs.open(path, flags, [mode], [callback(err, fd)])是 POSIX open 函数的封装，与 C 语言标准库中的 fopen 函数类似。它接受两个必选参数，path 为文件的路径，flags 可以是以下值。

- r ：以读取模式打开文件。
- r+：以读写模式打开文件。
- w ：以写入模式打开文件，如果文件不存在则创建。
- w+：以读写模式打开文件，如果文件不存在则创建。
- a ：以追加模式打开文件，如果文件不存在则创建。
- a+：以读取追加模式打开文件，如果文件不存在则创建。

mode 参数用于创建文件时给文件指定权限，默认是 0666[①]。回调函数将会传递一个文件描述符 fd[②]。

① 文件权限指的是 POSIX 操作系统中对文件读取和访问权限的规范，通常用一个八进制数来表示。例如 0754 表示文件所有者的权限是 7（读、写、执行），同组的用户权限是 5（读、执行），其他用户的权限是 4（读），写成字符表示就是 -rwxr-xr--。

② 文件描述符是一个非负整数，表示操作系统内核为当前进程所维护的打开文件的记录表索引。

4.4.4 fs.read

fs.read(fd, buffer, offset, length, position, [callback(err, bytesRead, buffer)])是 POSIX read 函数的封装，相比 fs.readFile 提供了更底层的接口。fs.read 的功能是从指定的文件描述符 fd 中读取数据并写入 buffer 指向的缓冲区对象。offset 是 buffer 的写入偏移量。length 是要从文件中读取的字节数。position 是文件读取的起始位置，如果 position 的值为 null，则会从当前文件指针的位置读取。回调函数传递 bytesRead 和 buffer，分别表示读取的字节数和缓冲区对象。

以下是一个使用 fs.open 和 fs.read 的示例。

```
var fs = require('fs');

fs.open('content.txt', 'r', function(err, fd) {
  if (err) {
    console.error(err);
    return;
  }

  var buf = new Buffer(8);
  fs.read(fd, buf, 0, 8, null, function(err, bytesRead, buffer) {
    if (err) {
      console.error(err);
      return;
    }

    console.log('bytesRead: ' + bytesRead);
    console.log(buffer);
  })
});
```

运行结果则是：

```
bytesRead: 8
<Buffer 54 65 78 74 20 e6 96 87>
```

一般来说，除非必要，否则不要使用这种方式读取文件，因为它要求你手动管理缓冲区和文件指针，尤其是在你不知道文件大小的时候，这将会是一件很麻烦的事情。

表4-1列出了fs所有函数的定义和功能。

表4-1　fs 模块函数表

功　能	异步函数	同步函数
打开文件	fs.open(path,flags, [mode], [callback(err, fd)])	fs.openSync(path, flags, [mode])
关闭文件	fs.close(fd, [callback(err)])	fs.closeSync(fd)

（续）

功　能	异步函数	同步函数
读取文件（文件描述符）	fs.read(fd,buffer,offset,length,position,[callback(err, bytesRead, buffer)])	fs.readSync(fd, buffer, offset, length, position)
写入文件（文件描述符）	fs.write(fd,buffer,offset,length,position,[callback(err, bytesWritten, buffer)])	fs.writeSync(fd, buffer, offset, length, position)
读取文件内容	fs.readFile(filename,[encoding],[callback(err, data)])	fs.readFileSync(filename, [encoding])
写入文件内容	fs.writeFile(filename, data,[encoding],[callback(err)])	fs.writeFileSync(filename, data, [encoding])
删除文件	fs.unlink(path, [callback(err)])	fs.unlinkSync(path)
创建目录	fs.mkdir(path, [mode], [callback(err)])	fs.mkdirSync(path, [mode])
删除目录	fs.rmdir(path, [callback(err)])	fs.rmdirSync(path)
读取目录	fs.readdir(path, [callback(err, files)])	fs.readdirSync(path)
获取真实路径	fs.realpath(path, [callback(err, resolvedPath)])	fs.realpathSync(path)
更名	fs.rename(path1, path2, [callback(err)])	fs.renameSync(path1, path2)
截断	fs.truncate(fd, len, [callback(err)])	fs.truncateSync(fd, len)
更改所有权	fs.chown(path, uid, gid, [callback(err)])	fs.chownSync(path, uid, gid)
更改所有权（文件描述符）	fs.fchown(fd, uid, gid, [callback(err)])	fs.fchownSync(fd, uid, gid)
更改所有权（不解析符号链接）	fs.lchown(path, uid, gid, [callback(err)])	fs.lchownSync(path, uid, gid)
更改权限	fs.chmod(path, mode, [callback(err)])	fs.chmodSync(path, mode)
更改权限（文件描述符）	fs.fchmod(fd, mode, [callback(err)])	fs.fchmodSync(fd, mode)
更改权限（不解析符号链接）	fs.lchmod(path, mode, [callback(err)])	fs.lchmodSync(path, mode)
获取文件信息	fs.stat(path, [callback(err, stats)])	fs.statSync(path)
获取文件信息（文件描述符）	fs.fstat(fd, [callback(err, stats)])	fs.fstatSync(fd)
获取文件信息（不解析符号链接）	fs.lstat(path, [callback(err, stats)])	fs.lstatSync(path)
创建硬链接	fs.link(srcpath, dstpath, [callback(err)])	fs.linkSync(srcpath, dstpath)
创建符号链接	fs.symlink(linkdata, path, [type], [callback(err)])	fs.symlinkSync(linkdata, path, [type])
读取链接	fs.readlink(path, [callback(err, linkString)])	fs.readlinkSync(path)
修改文件时间戳	fs.utimes(path, atime, mtime, [callback(err)])	fs.utimesSync(path, atime, mtime)
修改文件时间戳（文件描述符）	fs.futimes(fd, atime, mtime, [callback(err)])	fs.futimesSync(fd, atime, mtime)
同步磁盘缓存	fs.fsync(fd, [callback(err)])	fs.fsyncSync(fd)

4.5　HTTP 服务器与客户端

Node.js 标准库提供了 `http` 模块，其中封装了一个高效的 HTTP 服务器和一个简易的 HTTP 客户端。`http.Server` 是一个基于事件的 HTTP 服务器，它的核心由 Node.js 下层 C++ 部分实现，而接口由 JavaScript 封装，兼顾了高性能与简易性。`http.request` 则是一个 HTTP 客户端工具，用于向 HTTP 服务器发起请求，例如实现 Pingback[①]或者内容抓取。

4.5.1　HTTP 服务器

`http.Server` 是 `http` 模块中的 HTTP 服务器对象，用 Node.js 做的所有基于 HTTP 协议的系统，如网站、社交应用甚至代理服务器，都是基于 `http.Server` 实现的。它提供了一套封装级别很低的 API，仅仅是流控制和简单的消息解析，所有的高层功能都要通过它的接口来实现。

我们在 3.1.3 节中使用 `http` 实现了一个服务器：

```
//app.js

var http = require('http');

http.createServer(function(req, res) {
  res.writeHead(200, {'Content-Type': 'text/html'});
  res.write('<h1>Node.js</h1>');
  res.end('<p>Hello World</p>');
}).listen(3000);

console.log("HTTP server is listening at port 3000.");
```

这段代码中，`http.createServer` 创建了一个 `http.Server` 的实例，将一个函数作为 HTTP 请求处理函数。这个函数接受两个参数，分别是请求对象（ `req` ）和响应对象（ `res` ）。在函数体内，`res` 显式地写回了响应代码 200 （表示请求成功），指定响应头为 `'Content-Type': 'text/html'`，然后写入响应体 `'<h1>Node.js</h1>'`，通过 `res.end` 结束并发送。最后该实例还调用了 `listen` 函数，启动服务器并监听 3000 端口。

1. `http.Server` 的事件

`http.Server` 是一个基于事件的 HTTP 服务器，所有的请求都被封装为独立的事件，开发者只需要对它的事件编写响应函数即可实现 HTTP 服务器的所有功能。它继承自 `EventEmitter`，提供了以下几个事件。

---

① Pingback 是博客系统中用来通知文章被他人引用的一种手段，例如 WordPress 会自动解析文章中的链接，发送 Pingback 以告知链接被引用。

- request：当客户端请求到来时，该事件被触发，提供两个参数 req 和res，分别是 http.ServerRequest 和 http.ServerResponse 的实例，表示请求和响应信息。
- connection：当 TCP 连接建立时，该事件被触发，提供一个参数 socket，为 net.Socket 的实例。connection 事件的粒度要大于 request，因为客户端在 Keep-Alive 模式下可能会在同一个连接内发送多次请求。
- close：当服务器关闭时，该事件被触发。注意不是在用户连接断开时。

除此之外还有 checkContinue、upgrade、clientError 事件，通常我们不需要关心，只有在实现复杂的 HTTP 服务器的时候才会用到。

在这些事件中，最常用的就是 request 了，因此 http 提供了一个捷径：http.createServer([requestListener])，功能是创建一个 HTTP 服务器并将 requestListener 作为 request 事件的监听函数，这也是我们前面例子中使用的方法。事实上它显式的实现方法是：


```
//httpserver.js

var http = require('http');

var server = new http.Server();
server.on('request', function(req, res) {
  res.writeHead(200, {'Content-Type': 'text/html'});
  res.write('<h1>Node.js</h1>');
  res.end('<p>Hello World</p>');
});
server.listen(3000);

console.log("HTTP server is listening at port 3000.");
```

2. http.ServerRequest

http.ServerRequest 是 HTTP 请求的信息，是后端开发者最关注的内容。它一般由 http.Server 的 request 事件发送，作为第一个参数传递，通常简称 request 或 req。ServerRequest 提供一些属性，表 4-2 中列出了这些属性。

HTTP 请求一般可以分为两部分：请求头（Request Header）和请求体（Requset Body）。以上内容由于长度较短都可以在请求头解析完成后立即读取。而请求体可能相对较长，需要一定的时间传输，因此 http.ServerRequest 提供了以下3个事件用于控制请求体传输。

- data：当请求体数据到来时，该事件被触发。该事件提供一个参数 chunk，表示接收到的数据。如果该事件没有被监听，那么请求体将会被抛弃。该事件可能会被调用多次。

- end：当请求体数据传输完成时，该事件被触发，此后将不会再有数据到来。
- close：用户当前请求结束时，该事件被触发。不同于 end，如果用户强制终止了传输，也还是调用close。

表4-2　ServerRequest 的属性

名　称	含　义
complete	客户端请求是否已经发送完成
httpVersion	HTTP 协议版本，通常是 1.0 或 1.1
method	HTTP 请求方法，如 GET、POST、PUT、DELETE 等
url	原始的请求路径，例如 /static/image/x.jpg 或 /user?name=byvoid
headers	HTTP 请求头
trailers	HTTP 请求尾（不常见）
connection	当前 HTTP 连接套接字，为 net.Socket 的实例
socket	connection 属性的别名
client	client 属性的别名

3. 获取 GET 请求内容

注意，http.ServerRequest 提供的属性中没有类似于 PHP 语言中的 $_GET 或 $_POST 的属性，那我们如何接受客户端的表单请求呢？由于 GET 请求直接被嵌入在路径中，URL是完整的请求路径，包括了 ? 后面的部分，因此你可以手动解析后面的内容作为 GET 请求的参数。Node.js 的 url 模块中的 parse 函数提供了这个功能，例如：

```
//httpserverrequestget.js

var http = require('http');
var url = require('url');
var util = require('util');

http.createServer(function(req, res) {
  res.writeHead(200, {'Content-Type': 'text/plain'});
  res.end(util.inspect(url.parse(req.url, true)));
}).listen(3000);
```

在浏览器中访问 http://127.0.0.1:3000/user?name=byvoid&email=byvoid@byvoid.com，我们可以看到浏览器返回的结果：

```
{ search: '?name=byvoid&email=byvoid@byvoid.com',
  query: { name: 'byvoid', email: 'byvoid@byvoid.com' },
  pathname: '/user',
```

```
  path: '/user?name=byvoid&email=byvoid@byvoid.com',
  href: '/user?name=byvoid&email=byvoid@byvoid.com' }
```

通过 url.parse[1]，原始的 path 被解析为一个对象，其中 query 就是我们所谓的 GET 请求的内容，而路径则是 pathname。

4. 获取 POST 请求内容

HTTP 协议 1.1 版本提供了8种标准的请求方法，其中最常见的就是 GET 和 POST。相比 GET 请求把所有的内容编码到访问路径中，POST 请求的内容全部都在请求体中。http.ServerRequest 并没有一个属性内容为请求体，原因是等待请求体传输可能是一件耗时的工作，譬如上传文件。而很多时候我们可能并不需要理会请求体的内容，恶意的 POST 请求会大大消耗服务器的资源。所以 Node.js 默认是不会解析请求体的，当你需要的时候，需要手动来做。让我们看看实现方法：


```
//httpserverrequestpost.js

var http = require('http');
var querystring = require('querystring');
var util = require('util');

http.createServer(function(req, res) {
  var post = '';

  req.on('data', function(chunk) {
    post += chunk;
  });

  req.on('end', function() {
    post = querystring.parse(post);
    res.end(util.inspect(post));
  });

}).listen(3000);
```

上面代码并没有在请求响应函数中向客户端返回信息，而是定义了一个 post 变量，用于在闭包中暂存请求体的信息。通过 req 的 data 事件监听函数，每当接受到请求体的数据，就累加到 post 变量中。在 end 事件触发后，通过 querystring.parse 将 post 解析为真正的 POST 请求格式，然后向客户端返回。

警告 不要在真正的生产应用中使用上面这种简单的方法来获取 POST 请求，因为它有严重的效率问题和安全问题，这只是一个帮助你理解的示例。

[1] url 模块的说明参见 http://nodejs.org/api/url.html。

5. http.ServerResponse

http.ServerResponse 是返回给客户端的信息，决定了用户最终能看到的结果。它也是由 http.Server 的 request 事件发送的，作为第二个参数传递，一般简称为 response 或 res。

http.ServerResponse 有三个重要的成员函数，用于返回响应头、响应内容以及结束请求。

- response.writeHead(statusCode, [headers])：向请求的客户端发送响应头。statusCode 是 HTTP 状态码，如 200 （请求成功）、404 （未找到）等。headers 是一个类似关联数组的对象，表示响应头的每个属性。该函数在一个请求内最多只能调用一次，如果不调用，则会自动生成一个响应头。
- response.write(data, [encoding])：向请求的客户端发送响应内容。data 是一个 Buffer 或字符串，表示要发送的内容。如果 data 是字符串，那么需要指定 encoding 来说明它的编码方式，默认是 utf-8。在 response.end 调用之前，response.write 可以被多次调用。
- response.end([data], [encoding])：结束响应，告知客户端所有发送已经完成。当所有要返回的内容发送完毕的时候，该函数*必须*被调用一次。它接受两个可选参数，意义和 response.write 相同。如果不调用该函数，客户端将永远处于等待状态。

4.5.2 HTTP 客户端

http 模块提供了两个函数 http.request 和 http.get，功能是作为客户端向 HTTP 服务器发起请求。

- http.request(options, callback) 发起 HTTP 请求。接受两个参数，option 是一个类似关联数组的对象，表示请求的参数，callback 是请求的回调函数。option 常用的参数如下所示。
  - host ：请求网站的域名或 IP 地址。
  - port ：请求网站的端口，默认 80。
  - method ：请求方法，默认是 GET。
  - path ：请求的相对于根的路径，默认是“/”。QueryString 应该包含在其中。例如 /search?query=byvoid。
  - headers ：一个关联数组对象，为请求头的内容。

  callback 传递一个参数，为 http.ClientResponse 的实例。

  http.request 返回一个 http.ClientRequest 的实例。

下面是一个通过 `http.request` 发送 POST 请求的代码：

```
//httprequest.js

var http = require('http');
var querystring = require('querystring');

var contents = querystring.stringify({
  name: 'byvoid',
  email: 'byvoid@byvoid.com',
  address: 'Zijing 2#, Tsinghua University',
});

var options = {
  host: 'www.byvoid.com',
  path: '/application/node/post.php',
  method: 'POST',
  headers: {
    'Content-Type': 'application/x-www-form-urlencoded',
    'Content-Length' : contents.length
  }
};

var req = http.request(options, function(res) {
  res.setEncoding('utf8');
  res.on('data', function (data) {
    console.log(data);
  });
});

req.write(contents);
req.end();
```

运行后结果如下：

```
array(3) {
  ["name"]=>
  string(6) "byvoid"
  ["email"]=>
  string(17) "byvoid@byvoid.com"
  ["address"]=>
  string(30) "Zijing 2#, Tsinghua University"
}
```

提示　不要忘了通过 `req.end()` 结束请求，否则服务器将不会收到信息。

- http.get(options, callback) http 模块还提供了一个更加简便的方法用于处理GET请求：http.get。它是 http.request 的简化版，唯一的区别在于http.get自动将请求方法设为了 GET 请求，同时不需要手动调用 req.end()。

```
//httpget.js

var http = require('http');

http.get({host: 'www.byvoid.com'}, function(res) {
  res.setEncoding('utf8');
  res.on('data', function (data) {
    console.log(data);
  });
});
```

1. http.ClientRequest

http.ClientRequest 是由 http.request 或 http.get 返回产生的对象，表示一个已经产生而且正在进行中的 HTTP 请求。它提供一个 response 事件，即 http.request 或 http.get 第二个参数指定的回调函数的绑定对象。我们也可以显式地绑定这个事件的监听函数：

```
//httpresponse.js

var http = require('http');

var req = http.get({host: 'www.byvoid.com'});

req.on('response', function(res) {
  res.setEncoding('utf8');
  res.on('data', function (data) {
    console.log(data);
  });
});
```

http.ClientRequest 像 http.ServerResponse 一样也提供了 write 和 end 函数，用于向服务器发送请求体，通常用于 POST、PUT 等操作。所有写结束以后必须调用 end 函数以通知服务器，否则请求无效。http.ClientRequest 还提供了以下函数。

- request.abort()：终止正在发送的请求。
- request.setTimeout(timeout, [callback])：设置请求超时时间，timeout 为毫秒数。当请求超时以后，callback 将会被调用。

此外还有request.setNoDelay([noDelay])、request.setSocketKeepAlive([enable],[initialDelay]) 等函数，具体内容请参见 Node.js 文档。

2. http.ClientResponse

http.ClientResponse 与 http.ServerRequest 相似，提供了三个事件 data、end 和 close，分别在数据到达、传输结束和连接结束时触发，其中 data 事件传递一个参数 chunk，表示接收到的数据。

http.ClientResponse 也提供了一些属性，用于表示请求的结果状态，参见表 4-3。

表4-3 ClientResponse 的属性

名　称	含　义
statusCode	HTTP 状态码，如 200、404、500
httpVersion	HTTP 协议版本，通常是 1.0 或 1.1
headers	HTTP 请求头
trailers	HTTP 请求尾（不常见）

http.ClientResponse 还提供了以下几个特殊的函数。

- response.setEncoding([encoding])：设置默认的编码，当 data 事件被触发时，数据将会以 encoding 编码。默认值是 null，即不编码，以 Buffer 的形式存储。常用编码为 utf8。
- response.pause()：暂停接收数据和发送事件，方便实现下载功能。
- response.resume()：从暂停的状态中恢复。

4.6 参考资料

- Node.js Manual & Documentation: http://nodejs.org/api/index.html。
- “Understanding process.nextTick()”: http://howtonode.org/understanding- process-next-tick。
- “揭秘Node.js事件”: http://www.grati.org/?p=318。

使用Node.js进行Web开发

第 5 章

阅读到这一章为止，你已经学习了许多知识，但还缺乏实战性的内容。本章，我们打算从零开始用 Node.js 实现一个微博系统，功能包括路由控制、页面模板、数据库访问、用户注册、登录、用户会话等内容。

我们会介绍 Express 框架、MVC 设计模式、ejs 模板引擎以及 MongoDB 数据库的操作。通过实战演练，你将会了解到网站开发的基本方法。本章涉及的代码较多，所有的代码均可以在www.byvoid.com/project/node找到，但你最好还是亲自输入这些代码。现在就让我们开始一起动手来实现一个微博网站吧。

5.1　准备工作

在开始动手之前，我们首先要大致知道 Node.js 实现网站的工作原理。Node.js 和 PHP、Perl、ASP、JSP 一样，目的都是实现动态网页，也就是说由服务器动态生成 HTML 页面。之所以要这么做，是因为静态 HTML 的可扩展性非常有限，无法与用户有效交互。同时如果有大量相似的内容，例如产品介绍页面，那么1000个产品就要1000个静态的 HTML 页面，维护这1000个页面简直是一场灾难，因此动态生成 HTML 页面的技术应运而生。

最早实现动态网页的方法是使用Perl [①]和 CGI。在 Perl 程序中输出 HTML 内容，由 HTTP 服务器调用 Perl 程序，将结果返回给客户端。这种方式在互联网刚刚兴起的 20 世纪 90 年代非常流行，几乎所有的动态网页都是这么做的。但问题在于如果 HTML 内容比较多，维护非常不方便。大概在 2000 年左右，以 ASP、PHP、JSP 的为代表的以模板为基础的语言出现了，这种语言的使用方法与 CGI 相反，是在以 HTML 为主的模板中插入程序代码[②]。这种方式在2002年前后非常流行，但它的问题是页面和程序逻辑紧密耦合，任何一个网站规模变大以后，都会遇到结构混乱，难以处理的问题。为了解决这种问题，以 MVC 架构为基础的平台逐渐兴起，著名的 Ruby on Rails、Django、Zend Framework 都是基于 MVC 架构的。

MVC（Model-View-Controller，模型–视图–控制器）是一种软件的设计模式，它最早是由 20 世纪 70 年代的 Smalltalk 语言提出的，即把一个复杂的软件工程分解为三个层面：模型、视图和控制器。

- 模型是对象及其数据结构的实现，通常包含数据库操作。
- 视图表示用户界面，在网站中通常就是 HTML 的组织结构。
- 控制器用于处理用户请求和数据流、复杂模型，将输出传递给视图。

我们称 PHP、ASP、JSP 为“模板为中心的架构”，表 5-1 是两种Web开发架构的一个对比。

① 是C++，任何语言都可以，Perl只是最常见的。

② 例如 ASP 的 `<% %>` 和 PHP 的 `<?php ?>` 标签，在这些标签内添加处理代码。

表5-1 Web 开发架构对比

特性	模板为中心架构	MVC 架构
页面产生方式	执行并替换标签中的语句	由模板引擎生成 HTML 页面
路径解析	对应到文件系统[①]	由控制器定义
数据访问	通过 SQL 语句查询或访问文件系统	对象关系模型
架构中心	脚本语言是静态 HTTP 服务器的扩展	静态 HTTP 服务器是脚本语言的补充
适用范围	小规模网站	大规模网站
学习难度	容易	较难

这两种架构都出自原始的 CGI，但不同之处是前者走了一条粗放扩张的发展路线，由于易学易用，在几年前应用较广，而随着互联网规模的扩大，后者优势逐渐体现，目前已经成为主流。

Node.js 本质上和 Perl 或 C++ 一样，都可以作为 CGI 扩展被调用，但它还可以跳过 HTTP 服务器，因为它本身就是。传统的架构中 HTTP 服务器的角色会由 Apache、Nginx、IIS 之类的软件来担任，而 Node.js 不需要[②]。Node.js 提供了 `http` 模块，它是由 C++ 实现的，性能可靠，可以直接应用到生产环境。图5-1 是一个简单的架构示意图。

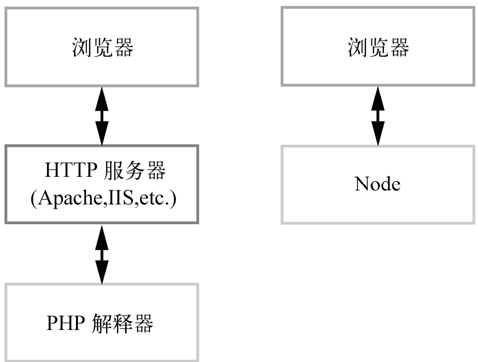


图5-1 Node.js 与 PHP架构的对比

Node.js 和其他的语言相比的另一个显著区别，在于它的原始封装程度较低。例如 PHP 中你可以访问 `$_REQUEST` 获取客户端的 POST 或 GET 请求，通常不需要直接处理 HTTP 协

① 例如 http://example.com/hello/world.php 对应服务器上的 /hello/world.php 这个文件。当然这不是绝对的，现在很多PHP开发框架都是只提供单个入口，利用服务器的 Rewrite 支持实现了路径的自由控制。我们一般情况下指的是原生的（或默认的）支持。

② 或者说不是必要的，因为你也可以把 Node.js 的服务器当作 Apache 或 Nginx 后端。

议[①]。这些语言要求由 HTTP 服务器来调用，因此你需要设置一个 HTTP 服务器来处理客户端的请求，HTTP 服务器通过 CGI 或其他方式调用脚本语言解释器，将运行的结果传递回 HTTP 服务器，最终再把内容返回给客户端。而在 Node.js 中，很多工作需要你自己来做（并不是都要自己动手，因为有第三方框架的帮助）。

5.1.1　使用 http 模块

Node.js 由于不需要另外的 HTTP 服务器，因此减少了一层抽象，给性能带来不少提升，但同时也因此而提高了开发难度。举例来说，我们要实现一个 POST 数据的表单，例如：

```
<form method="post" action="http://localhost:3000/">
<input type="text" name="title" />
<textarea name="text"></textarea>
<input type="submit" />
</form>
```

这个表单包含两个字段：title 和 text，提交时以 POST 的方式将请求发送给 http://localhost:3000/。假设我们要实现的功能是将这两个字段的东西原封不动地返回给用户，PHP 只需写两行代码，储存为 index.php 放在网站根目录下即可：

```
echo $_POST['title'];
echo $_POST['text'];
```

在 3.5.1 节中使用了类似下面的方法（用http模块）：

```
var http = require('http');
var querystring = require('querystring');

var server = http.createServer(function(req, res) {
  var post = '';

  req.on('data', function(chunk) {
    post += chunk;
  });

  req.on('end', function() {
    post = querystring.parse(post);

    res.write(post.title);
    res.write(post.text);
    res.end();
  });

}).listen(3000);
```

① 比如我们需要知道 HTTP 成功响应时要返回一个 200 状态码，而不需要手动完成“返回 200 状态码”这项工作。但这不带表你可以轻易地切换到非 HTTP 协议，因为代码仍然是与 HTTP 协议耦合的。

这种差别可能会让你大吃一惊，PHP 的实现要比Node.js容易得多。Node.js 完成这样一个简单任务竟然如此复杂：你需要先创建一个 `http` 的实例，在其请求处理函数中手动编写 `req` 对象的事件监听器。当客户端数据到达时，将 POST 数据暂存在闭包的变量中，直到 `end` 事件触发，解析 POST 请求，处理后返回客户端。

其实这个比较是不公平的，PHP 之所以显得简单并不是因为它没有做这些事，而是因为 PHP 已经将这些工作完全封装好了，只提供了一个高层的接口，而 Node.js 的 `http` 模块提供的是底层的接口，尽管使用起来复杂，却可以让我们对 HTTP 协议的理解更加清晰。

但是等等，我们并不是为了理解 HTTP 协议才来使用 Node.js 的，作为 Web 应用开发者，我们不需要知道实现的细节，更不想与这些细节纠缠从而降低开发效率。难道 Node.js 的抽象如此之差，把不该有的细节都暴露给了开发者吗？

实际上，Node.js 虽然提供了 `http` 模块，却不是让你直接用这个模块进行 Web 开发的。`http` 模块仅仅是一个 HTTP 服务器内核的封装，你可以用它做任何 HTTP 服务器能做的事情，不仅仅是做一个网站，甚至实现一个 HTTP 代理服务器都行。你如果想用它直接开发网站，那么就必须手动实现所有的东西了，小到一个 POST 请求，大到 Cookie、会话的管理。当你用这种方式建成一个网站的时候，你就几乎已经做好了一个完整的框架了。

5.1.2 Express 框架

npm 提供了大量的第三方模块，其中不乏许多 Web 框架，我们没有必要重复发明轮子，因而选择使用 Express 作为开发框架，因为它是目前最稳定、使用最广泛，而且 Node.js 官方推荐的唯一一个 Web 开发框架。

Express（ http://expressjs.com/ ）除了为 `http` 模块提供了更高层的接口外，还实现了许多功能，其中包括：

- 路由控制；
- 模板解析支持；
- 动态视图；
- 用户会话；
- CSRF 保护；
- 静态文件服务；
- 错误控制器；
- 访问日志；
- 缓存；
- 插件支持。

需要指出的是，Express 不是一个无所不包的全能框架，像 Rails 或 Django 那样实现了模板引擎甚至 ORM（Object Relation Model，对象关系模型）。它只是一个轻量级的 Web 框

架，多数功能只是对 HTTP 协议中常用操作的封装，更多的功能需要插件或者整合其他模块来完成。

下面用 Express 重新实现前面的例子：

```
var express = require('express');

var app = express.createServer();
app.use(express.bodyParser());
app.all('/', function(req, res) {
  res.send(req.body.title + req.body.text);
});

app.listen(3000);
```

可以看到，我们不需要手动编写 `req` 的事件监听器了，只需加载 `express.bodyParser()` 就能直接通过 `req.body` 获取 POST 的数据了。

5.2　快速开始

在上一小节我们已经介绍了 Web 开发的典型架构，我们选择了用 Express 作为开发框架来开发一个网站，从现在开始我们就要真正动手实践了。

5.2.1　安装 Express

首先我们要安装 Express。如果一个包是某个工程依赖，那么我们需要在工程的目录下使用本地模式安装这个包，如果要通过命令行调用这个包中的命令，则需要用全局模式安装（关于本地模式和全局模式，参见 3.3.4节），因此按理说我们使用本地模式安装 Express 即可。但是Express 像很多框架一样都提供了 Quick Start（快速开始）工具，这个工具的功能通常是建立一个网站最小的基础框架，在此基础上完成开发。当然你可以完全自己动手，但我还是推荐使用这个工具更快速地建立网站。为了使用这个工具，我们需要用全局模式安装 Express，因为只有这样我们才能在命令行中使用它。运行以下命令：

```
$ npm install -g express
```

等待数秒后安装完成，我们就可以在命令行下通过 `express` 命令快速创建一个项目了。在这之前先使用 `express --help` 查看帮助信息：

```
Usage: express [options] [path]

Options:
  -s, --sessions           add session support
  -t, --template <engine>  add template <engine> support (jade|ejs). default=jade
```

```
  -c, --css <engine>     add stylesheet <engine> support (stylus). default=plain css
  -v, --version          output framework version
  -h, --help             output help information
```

Express 在初始化一个项目的时候需要指定模板引擎，默认支持Jade和ejs，为了降低学习难度我们推荐使用 ejs ①，同时暂时不添加 CSS 引擎和会话支持。

5.2.2 建立工程

通过以下命令建立网站基本结构：

```
express -t ejs microblog
```

当前目录下出现了子目录 microblog，并且产生了一些文件：

```
   create : microblog
   create : microblog/package.json
   create : microblog/app.js
   create : microblog/public
   create : microblog/public/javascripts
   create : microblog/public/images
   create : microblog/public/stylesheets
   create : microblog/public/stylesheets/style.css
   create : microblog/routes
   create : microblog/routes/index.js
   create : microblog/views
   create : microblog/views/layout.ejs
   create : microblog/views/index.ejs

dont forget to install dependencies:
$ cd microblog && npm install
```

它还提示我们要进入其中运行 `npm install`，我们依照指示，结果如下：

```
ejs@0.6.1 ./node_modules/ejs
express@2.5.8 ./node_modules/express
-- qs@0.4.2
-- mime@1.2.4
-- mkdirp@0.3.0
-- connect@1.8.5
```

它自动安装了依赖 ejs 和 express。这是为什么呢？检查目录中的 package.json 文件，内容是：

① ejs（Embedded JavaScript）是一个标签替换引擎，其语法与 ASP、PHP 相似，易于学习，目前被广泛应用。Express 默认提供的引擎是 jade，它颠覆了传统的模板引擎，制定了一套完整的语法用来生成 HTML 的每个标签结构，功能强大但不易学习。

```
{
    "name": "microblog"
  , "version": "0.0.1"
  , "private": true
  , "dependencies": {
      "express": "2.5.8"
    , "ejs": ">= 0.0.1"
  }
}
```

其中 `dependencies` 属性中有`express` 和`ejs`。无参数的 `npm install` 的功能就是检查当前目录下的 package.json，并自动安装所有指定的依赖。

5.2.3　启动服务器

用 Express 实现的网站实际上就是一个 Node.js 程序，因此可以直接运行。我们运行 `node app.js`，看到 `Express server listening on port 3000 in development mode`。

接下来，打开浏览器，输入地址 http://localhost:3000，你就可以看到一个简单的 Welcome to Express 页面了。如果你能看到如图5-2 所示的页面，那么说明你的设定正确无误。

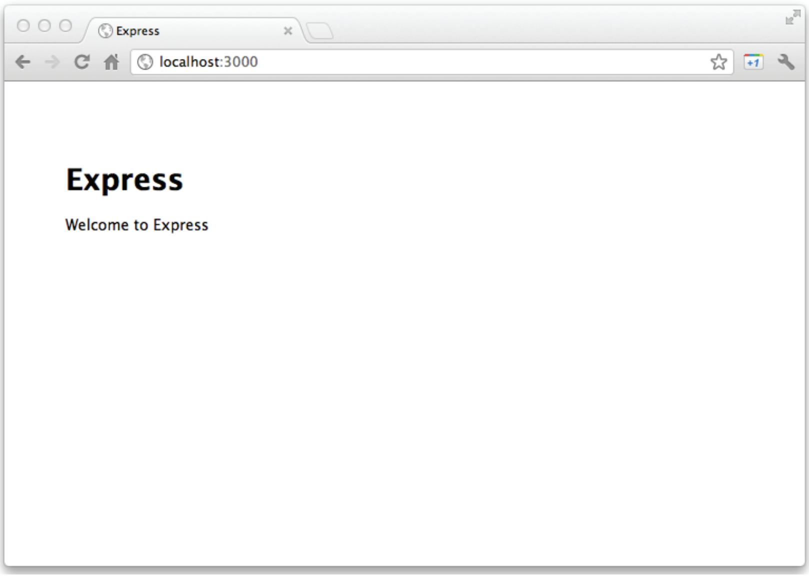


图5-2　Express 初始欢迎页面

要关闭服务器的话，在终端中按 Ctrl + C。注意，如果你对代码做了修改，要想看到修改后的效果必须重启服务器，也就是说你需要关闭服务器并再次运行才会有效果。如果觉得有些麻烦，可以使用 supervisor 实现监视代码修改和自动重启，具体使用方法参见 3.1.3 节。

提示

注意命令行中显示服务器运行在开发模式下（development mode），因此不要在生产环境中部署它。我们会在 6.3 节中介绍如何在真实的生产环境下部署 Node.js 服务器。

5.2.4 工程的结构

现在让我们回过头来看看 Express 都生成了哪些文件。除了 package.json，它只产生了两个 JavaScript 文件 app.js 和 routes/index.js。模板引擎 ejs 也有两个文件 index.ejs 和layout.ejs，此外还有样式表 style.css。下面来详细看看这几个文件。

1. app.js

app.js 是工程的入口，我们先看看其中有什么内容：

```
/**
 * Module dependencies.
 */

var express = require('express')
  , routes = require('./routes');

var app = module.exports = express.createServer();

// Configuration

app.configure(function(){
  app.set('views', __dirname + '/views');
  app.set('view engine', 'ejs');
  app.use(express.bodyParser());
  app.use(express.methodOverride());
  app.use(app.router);
  app.use(express.static(__dirname + '/public'));
});

app.configure('development', function(){
  app.use(express.errorHandler({ dumpExceptions: true, showStack: true }));
});

app.configure('production', function(){
  app.use(express.errorHandler());
});

// Routes

app.get('/', routes.index);

app.listen(3000);
console.log("Express server listening on port %d in %s mode", app.address().port,
app.settings.env);
```

5

对比上一节使用 Express 的例子，这个文件长了不少，不过并不复杂。下面来分析一下这段代码。

首先我们导入了 Express 模块，前面已经通过 npm 安装到了本地，在这里可以直接通过 `require` 获取。`routes` 是一个文件夹形式的本地模块，即 `./routes/index.js`，它的功能是为指定路径组织返回内容，相当于 MVC 架构中的控制器。通过 `express.createServer()` 函数创建了一个应用的实例，后面的所有操作都是针对于这个实例进行的。

接下来是三个 `app.configure` 函数，分别指定了通用、开发和产品环境下的参数。第一个 `app.configure` 直接接受了一个回调函数，后两个则只能在开发和产品环境中调用。

`app.set` 是 Express 的参数设置工具，接受一个键（key）和一个值（value），可用的参数如下所示。

- `basepath`：基础地址，通常用于 `res.redirect()` 跳转。
- `views`：视图文件的目录，存放模板文件。
- `view engine`：视图模板引擎。
- `view options`：全局视图参数对象。
- `view cache`：启用视图缓存。
- `case sensitive routes`：路径区分大小写。
- `strict routing`：严格路径，启用后不会忽略路径末尾的“ / ”。
- `jsonp callback`：开启透明的 JSONP 支持。

Express 依赖于 connect，提供了大量的中间件，可以通过 `app.use` 启用。`app.configure` 中启用了5个中间件：`bodyParser`、`methodOverride`、`router`、`static` 以及 `errorHandler`。`bodyParser` 的功能是解析客户端请求，通常是通过 POST 发送的内容。`methodOverride` 用于支持定制的 HTTP 方法[①]。`router` 是项目的路由支持。`static` 提供了静态文件支持。`errorHandler` 是错误控制器。

`app.get('/', routes.index);` 是一个路由控制器，用户如果访问“ / ”路径，则由 `routes.index` 来控制。

最后服务器通过 `app.listen(3000);` 启动，监听3000端口。

2. routes/index.js

routes/index.js 是路由文件，相当于控制器，用于组织展示的内容：

```
/*
 * GET home page.
 */

exports.index = function(req, res) {
```

① 如PUT、DELETE等HTTP方法，浏览器是不支持的。

```
  res.render('index', { title: 'Express' });
};
```

app.js 中通过 app.get('/', routes.index); 将“/”路径映射到 exports.index 函数下。其中只有一个语句 res.render('index', { title: 'Express' })，功能是调用模板解析引擎，翻译名为 index 的模板，并传入一个对象作为参数，这个对象只有一个属性，即 title: 'Express'。

3. index.ejs

index.ejs 是模板文件，即 routes/index.js 中调用的模板，内容是：

```
<h1><%= title %></h1>
<p>Welcome to <%= title %></p>
```

它的基础是 HTML 语言，其中包含了形如 <%= title %> 的标签，功能是显示引用的变量，即 res.render 函数第二个参数传入的对象的属性。

4. layout.ejs

模板文件不是孤立展示的，默认情况下所有的模板都继承自 layout.ejs，即 <%- body %> 部分才是独特的内容，其他部分是共有的，可以看作是页面框架。


```
<!DOCTYPE html>
<html>
  <head>
    <title><%= title %></title>
    <link rel='stylesheet' href='/stylesheets/style.css' />
  </head>
  <body>
    <%- body %>
  </body>
</html>
```

以上就是一个基本的工程结构，十分简单，功能划分却非常清楚。我们会在后面的小节中基于这个工程继续完善，直到实现一个功能完整的网站。

5.3 路由控制

在上一节，我们已经讲过了如何使用 Express 建立一个基本工程，这个工程只包含一些基础架构，没有任何实际内容。从这一小节开始，我们将会讲述 Express 的基本使用方法，在前面例子的基础上逐步完善这个工程。

5.3.1 工作原理

当通过浏览器访问 app.js 建立的服务器时，会看到一个简单的页面，实际上它已经

完成了许多透明的工作，现在就让我们来解释一下它的工作机制，以帮助理解网站的整体架构。

访问 http://localhost:3000，浏览器会向服务器发送以下请求：

```
GET / HTTP/1.1
Host: localhost:3000
Connection: keep-alive
Cache-Control: max-age=0
User-Agent: Mozilla/5.0 AppleWebKit/535.19 (KHTML, like Gecko) Chrome/18.0.1025.142
  Safari/535.19
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Encoding: gzip,deflate,sdch
Accept-Language: zh;q=0.8,en-US;q=0.6,en;q=0.4
Accept-Charset: UTF-8,*;q=0.5
```

其中第一行是请求的方法、路径和 HTTP 协议版本，后面若干行是 HTTP 请求头。`app` 会解析请求的路径，调用相应的逻辑。app.js 中有一行内容是 `app.get('/', routes.index)`，它的作用是规定路径为“/”的 GET 请求由 `routes.index` 函数处理。`routes.index` 通过 `res.render('index', { title: 'Express' })` 调用视图模板 `index`，传递 `title` 变量。最终视图模板生成 HTML 页面，返回给浏览器，返回的内容是：

```
HTTP/1.1 200 OK
X-Powered-By: Express
Content-Type: text/html; charset=utf-8
Content-Length: 202
Connection: keep-alive

<!DOCTYPE html>
<html>
  <head>
    <title>Express</title>
    <link rel='stylesheet' href='/stylesheets/style.css' />
  </head>
  <body>
    <h1>Express</h1>
<p>Welcome to Express</p>
  </body>
</html>
```

浏览器在接收到内容以后，经过分析发现要获取 /stylesheets/style.css，因此会再次向服务器发起请求。app.js 中并没有一个路由规则指派到 /stylesheets/style.css，但 `app` 通过 `app.use(express.static(__dirname + '/public'))` 配置了静态文件服务器，因此 /stylesheets/style.css 会定向到 app.js 所在目录的子目录中的文件 public/stylesheets/style.css，向客户端返回以下信息：

```
HTTP/1.1 200 OK
X-Powered-By: Express
Date: Mon, 02 Apr 2012 15:56:55 GMT
Cache-Control: public, max-age=0
Last-Modified: Mon, 12 Mar 2012 12:49:50 GMT
ETag: "110-1331556590000"
Content-Type: text/css; charset=UTF-8
Accept-Ranges: bytes
Content-Length: 110
Connection: keep-alive

body {
  padding: 50px;
  font: 14px "Lucida Grande", Helvetica, Arial, sans-serif;
}

a {
  color: #00B7FF;
}
```

由 Express 创建的网站架构如图5-3 所示。

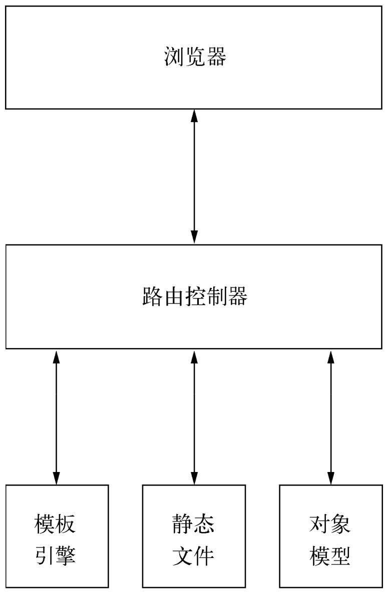


图5-3 Express 网站的架构

这是一个典型的 MVC 架构，浏览器发起请求，由路由控制器接受，根据不同的路径定向到不同的控制器。控制器处理用户的具体请求，可能会访问数据库中的对象，即模型部

分。控制器还要访问模板引擎，生成视图的 HTML，最后再由控制器返回给浏览器，完成一次请求。

5.3.2　创建路由规则

当我们在浏览器中访问譬如 http://localhost:3000/abc 这样不存在的页面时，服务器会在响应头中返回 404 Not Found 错误，浏览器显示如图5-4 所示。

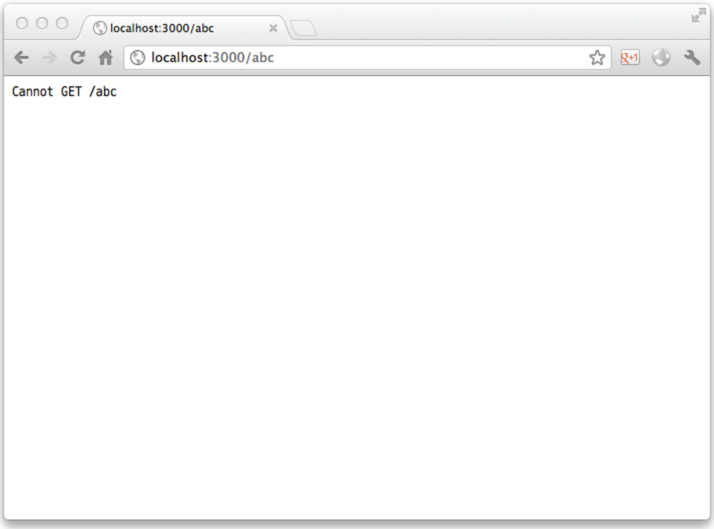


图5-4　访问不存在的页面时浏览器看到的结果

这是因为 /abc 是一个不存在的路由规则，而且它也不是一个 public 目录下的文件，所以Express返回了404 Not Found的错误。

接下来我们会讲述如何创建路由规则。

假设我们要创建一个地址为 `/hello` 的页面，内容是当前的服务器时间，让我们看看具体做法。打开 app.js，在已有的路由规则 `app.get('/', routes.index)` 后面添加一行：

```
app.get('/hello', routes.hello);
```

修改 routes/index.js，增加 `hello` 函数：

```
/*
 * GET home page.
 */

exports.index = function(req, res) {
  res.render('index', { title: 'Express' });
```

```
};

exports.hello = function(req, res) {
  res.send('The time is ' + new Date().toString());
};
```

重启 app.js，在浏览器中访问 http://localhost:3000/hello，可以看到类似于图5-5 的页面，刷新页面可以看到时间发生变化，因为你看到的内容是动态生成的结果。

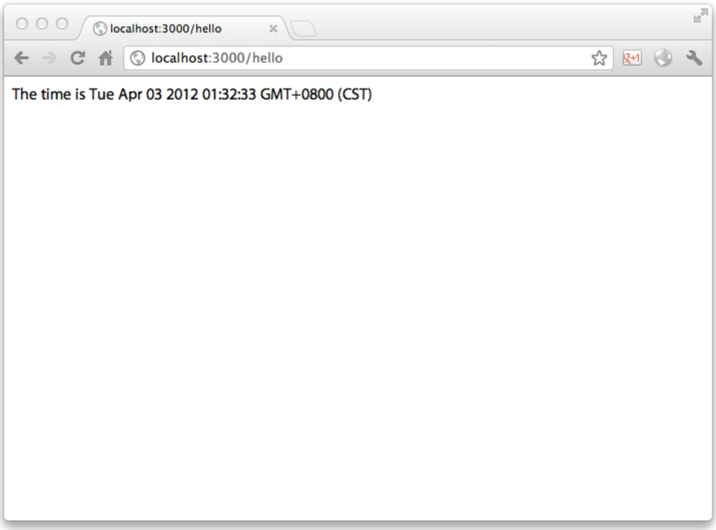


图5-5 访问 /hello 时显示的内容

服务器在开始监听之前，设置好了所有的路由规则，当请求到达时直接分配到响应函数。app.get 是路由规则创建函数，它接受两个参数，第一个参数是请求的路径，第二个参数是一个回调函数，该路由规则被触发时调用回调函数，其参数表传递两个参数，分别是 req 和 res，表示请求信息和响应信息。

5.3.3 路径匹配

上面的例子是为固定的路径设置路由规则，Express 还支持更高级的路径匹配模式。例如我们想要展示一个用户的个人页面，路径为 /user/[username]，可以用下面的方法定义路由规则：

```
app.get('/user/:username', function(req, res) {
  res.send('user: ' + req.params.username);
});
```

修改以后重启 app.js，访问 http://localhost:3000/user/byvoid，可以看到页面显示了以下内容：

```
user: byvoid
```

路径规则 /user/:username 会被自动编译为正则表达式，类似于 `\/user\/([^\/]+)\/?` 这样的形式。路径参数可以在响应函数中通过 `req.params` 的属性访问。

路径规则同样支持 JavaScript 正则表达式，例如 `app.get(\/user\/([^\/]+)\/?, callback)`。这样的好处在于可以定义更加复杂的路径规则，而不同之处是匹配的参数是匿名的，因此需要通过 `req.params[0]`、`req.params[1]` 这样的形式访问。

5.3.4　REST 风格的路由规则

Express 支持 REST 风格的请求方式，在介绍之前我们先说明一下什么是 REST。REST 的意思是 *表征状态转移*（Representational State Transfer），它是一种基于 HTTP 协议的网络应用的接口风格，充分利用 HTTP 的方法实现统一风格接口的服务。HTTP 协议定义了以下8种标准的方法。

- GET：请求获取指定资源。
- HEAD：请求指定资源的响应头。
- POST：向指定资源提交数据。
- PUT：请求服务器存储一个资源。
- DELETE：请求服务器删除指定资源。
- TRACE：回显服务器收到的请求，主要用于测试或诊断。
- CONNECT：HTTP/1.1 协议中预留给能够将连接改为管道方式的代理服务器。
- OPTIONS：返回服务器支持的HTTP请求方法。

其中我们经常用到的是 GET、POST、PUT 和 DELETE 方法。根据 REST 设计模式，这4种方法通常分别用于实现以下功能。

- GET：获取
- POST：新增
- PUT：更新
- DELETE：删除

这是因为这4种方法有不同的特点，按照定义，它们的特点如表 5-2 所示。

所谓安全是指没有副作用，即请求不会对资源产生变动，连续访问多次所获得的结果不受访问者的影响。而幂等指的是重复请求多次与一次请求的效果是一样的，比如获取和更新操作是幂等的，这与新增不同。删除也是幂等的，即重复删除一个资源，和删除一次是一样的。

表5-2 REST风格HTTP 请求的特点

请求方式	安 全	幂 等
GET	是	是
POST	否	否
PUT	否	是
DELETE	否	是

Express 对每种 HTTP 请求方法都设计了不同的路由绑定函数，例如前面例子全部是 `app.get`，表示为该路径绑定了 GET 请求，向这个路径发起其他方式的请求不会被响应。表 5-3 是 Express 支持的所有 HTTP 请求的绑定函数。

表5-3 Express 支持的 HTTP 请求的绑定函数

请求方式	绑定函数
GET	`app.get(path, callback)`
POST	`app.post(path, callback)`
PUT	`app.put(path, callback)`
DELETE	`app.delete(path, callback)`
PATCH[①]	`app.patch(path, callback)`
TRACE	`app.trace(path, callback)`
CONNECT	`app.connect(path, callback)`
OPTIONS	`app.options(path, callback)`
所有方法	`app.all(path, callback)`

例如我们要绑定某个路径的 POST 请求，则可以用 `app.post(path, callback)` 的方法。需要注意的是 `app.all` 函数，它支持把所有的请求方式绑定到同一个响应函数，是一个非常灵活的函数，在后面我们可以看到许多功能都可以通过它来实现。

5.3.5 控制权转移

Express 支持同一路径绑定多个路由响应函数，例如：

```
app.all('/user/:username', function(req, res) {
  res.send('all methods captured');
});
app.get('/user/:username', function(req, res) {
  res.send('user: ' + req.params.username);
});
```

① PATCH 方式是 IETF RFC 5789 （ http://tools.ietf.org/html/rfc5789 ） 新增的 HTTP 方法，功能定义是部分更新某个资源。

但当你访问任何被这两条同样的规则匹配到的路径时，会发现请求总是被前一条路由规则捕获，后面的规则会被忽略。原因是 Express 在处理路由规则时，会优先匹配先定义的路由规则，因此后面相同的规则被屏蔽。

Express 提供了路由控制权转移的方法，即回调函数的第三个参数`next`，通过调用`next()`，会将路由控制权转移给后面的规则，例如：

```
app.all('/user/:username', function(req, res, next) {
  console.log('all methods captured');
  next();
});
app.get('/user/:username', function(req, res) {
  res.send('user: ' + req.params.username);
});
```

当访问被匹配到的路径时，如 http://localhost:3000/user/carbo，会发现终端中打印了`all methods captured`，而且浏览器中显示了`user: carbo`。这说明请求先被第一条路由规则捕获，完成`console.log`使用`next()`转移控制权，又被第二条规则捕获，向浏览器返回了信息。

这是一个非常有用的工具，可以让我们轻易地实现中间件，而且还能提高代码的复用程度。例如我们针对一个用户查询信息和修改信息的操作，分别对应了 GET 和 PUT 操作，而两者共有的一个步骤是检查用户名是否合法，因此可以通过`next()`方法实现：

```
var users = {
  'byvoid': {
    name: 'Carbo',
    website: 'http://www.byvoid.com'
  }
};

app.all('/user/:username', function(req, res, next) {
  // 检查用户是否存在
  if (users[req.params.username]) {
    next();
  } else {
    next(new Error(req.params.username + ' does not exist.'));
  }
});
app.get('/user/:username', function(req, res) {
  // 用户一定存在，直接展示
  res.send(JSON.stringify(users[req.params.username]));
});
app.put('/user/:username', function(req, res) {
  // 修改用户信息
  res.send('Done');
});
```

上面例子中，`app.all` 定义的这个路由规则实际上起到了中间件的作用，把相似请求的相同部分提取出来，有利于代码维护其他`next`方法如果接受了参数，即代表发生了错误。使用这种方法可以把错误检查分段化，降低代码耦合度。

5.4 模板引擎

上一节我们介绍了 Express 的路由控制方法，它是网站架构最核心的部分，即MVC架构中的控制器。在这一小节里，我们会讲述模板引擎的使用和集成，也就是视图。视图决定了用户最终能看到什么，因此也是最重要部分，这里我们以 ejs 为例介绍模板引擎的使用方法。

5.4.1 什么是模板引擎

模板引擎（Template Engine）是一个从页面模板根据一定的规则生成 HTML 的工具。它的发轫可以追溯到 1996 年 PHP 2.0 的诞生。PHP 原本是 Personal Home Page Tools（个人主页工具）的简称，用于取代 Perl 和 CGI 的组合，其功能是让代码嵌入在 HTML 中执行，以产生动态的页面，因此 PHP 堪称是最早的模板引擎的雏形。随后的 ASP、JSP 都沿用了这个模式，即建立一个 HTML 页面模板，插入可执行的代码，运行时动态生成 HTML。

5

按照这种模式，整个网站就由一个个的页面模板组成，所有的逻辑都嵌入在模板中。这种模式大大降低了动态网页开发的门槛，因此一开始很受欢迎，但随着规模的扩大它会遇到许多问题，下面列举几个主要的。

- 页面功能逻辑与页面布局样式耦合，网站规模变大以后逐渐难以维护。
- 语法复杂，对于非技术的网页设计者来说门槛较高，难以学习。
- 功能过于全面，页面设计者可以在页面上编程，不利于功能划分，也使模板解析效率降低。

这些问题制约了早期模板引擎的发展，直到 MVC 开发模式普及，模板引擎才开始遍地开花。现代的模板引擎是 MVC 的一部分，在功能划分上它严格属于视图部分，因此功能以生成 HTML 页面为核心，不会引入过多的编程语言的功能。相较于一门编程语言，它通常学习起来相当容易。

模板引擎的功能是将页面模板和要显示的数据结合起来生成 HTML 页面。它既可以运行在服务器端又可以运行在客户端，大多数时候它都在服务器端直接被解析为 HTML，解析完成后再传输给客户端，因此客户端甚至无法判断页面是否是模板引擎生成的。有时候模板引擎也可以运行在客户端，即浏览器中，典型的代表就是 XSLT，它以 XML 为输入，在客户端生成 HTML 页面。但是由于浏览器兼容性问题，XSLT 并不是很流行。目前的主流还是由服务器运行模板引擎。

在 MVC 架构中，模板引擎包含在服务器端。控制器得到用户请求后，从模型获取数据，

调用模板引擎。模板引擎以数据和页面模板为输入，生成 HTML 页面，然后返回给控制器，由控制器交回客户端。图5-6 是模板引擎在 MVC 架构中的示意图。

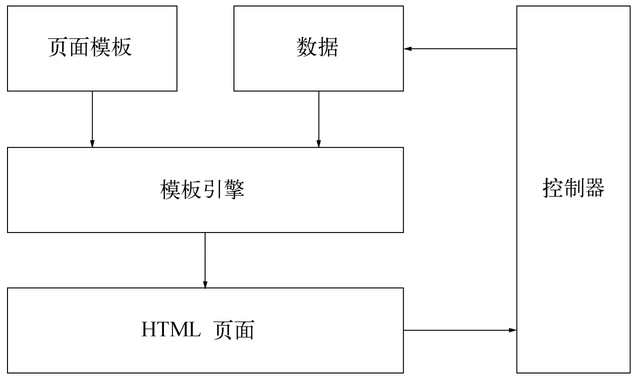


图5-6　模板引擎在 MVC 架构中的位置

5.4.2　使用模板引擎

基于 JavaScript 的模板引擎有许多种实现，我们推荐使用 ejs（Embedded JavaScript），因为它十分简单，而且与 Express 集成良好。由于它是标准 JavaScript 实现的，因此它不仅可以运行在服务器端，还可以运行在浏览器中。我们这一章的示例是在服务器端运行 ejs，这样减少了对浏览器的依赖，而且更符合传统架构的习惯。

我们在 app.js 中通过以下两个语句设置了模板引擎和页面模板的位置：

```
app.set('views', __dirname + '/views');
app.set('view engine', 'ejs');
```

表明要使用的模板引擎是 ejs，页面模板在 views 子目录下。在 routes/index.js 的 `exports.index` 函数中通过如下语句调用模板引擎：

```
res.render('index', { title: 'Express' });
```

`res.render` 的功能是调用模板引擎，并将其产生的页面直接返回给客户端。它接受两个参数，第一个是模板的名称，即 views 目录下的模板文件名，不包含文件的扩展名；第二个参数是传递给模板的数据，用于模板翻译。index.ejs 内容如下：

```
<h1><%= title %></h1>
<p>Welcome to <%= title %></p>
```

上面代码其中有两处 `<%= title %>`，用于模板变量显示，它们在模板翻译时会被替换成 Express，因为 `res.render` 传递了 `{ title: 'Express' }`。

ejs 的标签系统非常简单，它只有以下3种标签。

- `<% code %>`：JavaScript 代码。
- `<%= code %>`：显示替换过 HTML 特殊字符的内容。
- `<%- code %>`：显示原始 HTML 内容。

我们可以用它们实现页面模板系统能实现的任何内容。

5.4.3　页面布局

上面的例子介绍了页面模板的翻译，但我们看到的不止这两行，原因是 Express 还自动套用了 layout.ejs，它的内容是：

```
<!DOCTYPE html>
<html>
  <head>
    <title><%= title %></title>
    <link rel='stylesheet' href='/stylesheets/style.css' />
  </head>
  <body>
    <%- body %>
  </body>
</html>
```

layout.ejs 是一个页面布局模板，它描述了整个页面的框架结构，默认情况下每个单独的页面都继承自这个框架，替换掉 `<%- body %>` 部分。这个功能通常非常有用，因为一般为了保持整个网站的一致风格，HTML 页面的`<head>`部分以及页眉页脚中的大量内容是重复的，因此我们可以把它们放在 layout.ejs 中。当然，这个功能并不是强制的，如果想关闭它，可以在 app.js 的中 `app.configure` 中添加以下内容，这样页面布局功能就被关闭了。

```
app.set('view options', {
  layout: false
});
```

另一种情况是，一个网站可能需要不止一种页面布局，例如网站分前台展示和后台管理系统，两者的页面结构有很大的区别，一套页面布局不能满足需求。这时我们可以在页面模板翻译时指定页面布局，即设置 `layout` 属性，例如：

```
function(req, res) {
  res.render('userlist', {
    title: '用户列表-后台管理系统',
    layout: 'admin'
  });
};
```

这段代码会在翻译 `userlist` 页面模板时套用 admin.ejs 作为页面布局。

5.4.4　片段视图

Express 的视图系统还支持片段视图（partials），它就是一个页面的片段，通常是重复的内容，用于迭代显示。通过它你可以将相对独立的页面块分割出去，而且可以避免显式地使用 for 循环。让我们看一个例子，在 app.js 中新增以下内容：

```
app.get('/list', function(req, res) {
  res.render('list', {
    title: 'List',
    items: [1991, 'byvoid', 'express', 'Node.js']
  });
});
```

在 views 目录下新建 list.ejs，内容是：

```
<ul><%- partial('listitem', items) %></ul>
```

同时新建 listitem.ejs，内容是：

```
<li><%= listitem %></li>
```

访问 http://localhost:3000/list，可以在源代码中看到以下内容：

```
<!DOCTYPE html>
<html>
  <head>
    <title>List</title>
    <link rel='stylesheet' href='/stylesheets/style.css' />
  </head>
  <body>
    <ul><li>1991</li><li>byvoid</li><li>express</li><li>Node.js</li></ul>
  </body>
</html>
```

partial 是一个可以在视图中使用函数，它接受两个参数，第一个是片段视图的名称，第二个可以是一个对象或一个数组，如果是一个对象，那么片段视图中上下文变量引用的就是这个对象；如果是一个数组，那么其中每个元素依次被迭代应用到片段视图。片段视图中上下文变量名就是视图文件名，例如上面的'listitem'。

5.4.5　视图助手

Express 提供了一种叫做视图助手的工具，它的功能是允许在视图中访问一个全局的函数或对象，不用每次调用视图解析的时候单独传入。前面提到的 partial 就是一个视图助手。

视图助手有两类，分别是静态视图助手和动态视图助手。这两者的差别在于，静态视图助手可以是任何类型的对象，包括接受任意参数的函数，但访问到的对象必须是与用户请求无关的，而动态视图助手只能是一个函数，这个函数不能接受参数，但可以访问 req 和 res 对象。

静态视图助手可以通过 app.helpers() 函数注册，它接受一个对象，对象的每个属性名称为视图助手的名称，属性值对应视图助手的值。动态视图助手则通过 app.dynamicHelpers() 注册，方法与静态视图助手相同，但每个属性的值必须为一个函数，该函数提供 req 和 res，参见下面这个示例：

```
var util = require('util');

app.helpers({
  inspect: function(obj) {
    return util.inspect(obj, true);
  }
});

app.dynamicHelpers({
  headers: function(req, res) {
    return req.headers;
  }
});

app.get('/helper', function(req, res) {
  res.render('helper', {
    title: 'Helpers'
  });
});
```


对应的视图helper、ejs的内容如下：

```
<%=inspect(headers)%>
```

访问 http://localhost:3000/helper 可以看到如图5-7 所示的内容。

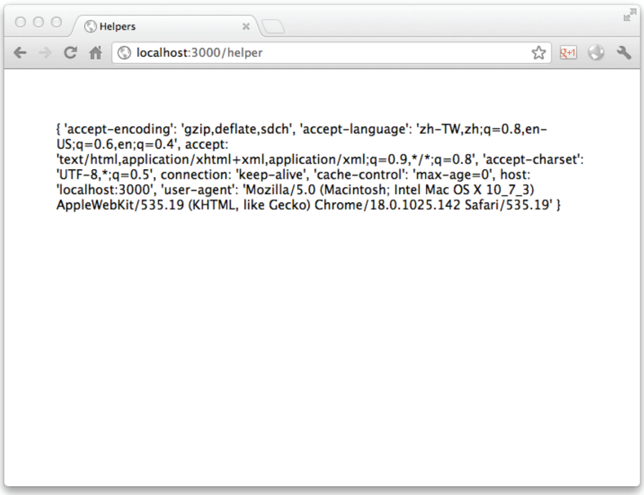


图5-7 使用视图助手的页面

视图助手的本质其实就是给所有视图注册了全局变量，因此无需每次在调用模板引擎时传递数据对象。当我们在后面使用 `session` 时会发现它是非常有用的。

5.5　建立微博网站

在前面的几节中，我们已经对 Express 进行了基本的介绍，现在让我们动手开始创建一个微博网站吧。

5.5.1　功能分析

开发中的一个大忌就是没有想清楚要做什么就开始动手，因此我们准备在动手实践之前先规划一下网站的功能，即使是出于学习目的也不例外。首先，微博应该以用户为中心，因此需要有用户的注册和登录功能。微博网站最核心的功能是信息的发表，这个功能涉及许多方面，包括数据库访问、前端显示等。一个完整的微博系统应该支持信息的评论、转发、圈点用户等功能，但出于演示目的，我们不能一一实现所有功能，只是实现一个微博社交网站的雏形。

5.5.2　路由规划

在完成功能设计以后，下一个要做的事情就是路由规划了。路由规划，或者说控制器规划是整个网站的骨架部分，因为它处于整个架构的枢纽位置，相当于各个接口之间的粘合剂，所以应该优先考虑。

根据功能设计，我们把路由按照以下方案规划。

- /：首页
- /u/[user]：用户的主页
- /post：发表信息
- /reg：用户注册
- /login：用户登录
- /logout：用户登出

以上页面还可以根据用户状态细分。发表信息以及用户登出页面必须是已登录用户才能操作的功能，而用户注册和用户登入所面向的对象必须是未登入的用户。首页和用户主页则针对已登入和未登入的用户显示不同的内容。

打开 app.js，把 `Routes` 部分修改为：

```
app.get('/', routes.index);
app.get('/u/:user', routes.user);
app.post('/post', routes.post);
```

```
app.get('/reg', routes.reg);
app.post('/reg', routes.doReg);
app.get('/login', routes.login);
app.post('/login', routes.doLogin);
app.get('/logout', routes.logout);
```

其中 /post、/login 和 /reg 由于要接受表单信息，因此使用 `app.post` 注册路由。/login 和 /reg 还要显示用户注册时要填写的表单，所以要以 `app.get` 注册。同时在 routes/index.js 中添加相应的函数：

```
exports.index = function(req, res) {
  res.render('index', { title: 'Express' });
};

exports.user = function(req, res) {
};

exports.post = function(req, res) {
};

exports.reg = function(req, res) {
};

exports.doReg = function(req, res) {
};

exports.login = function(req, res) {
};

exports.doLogin = function(req, res) {
};

exports.logout = function(req, res) {
};
```

我们将在5.6节介绍会话（session），说明如何管理用户的状态。

5.5.3 界面设计

我们在开发网站的时候必须时刻意识到网站是为用户开发的，因而用户界面是非常重要的。一种普遍的观点是后端的开发者不必太多关注前端用户体验，因为这是前端程序员和设计师要做的事情。但实际上为了设计一个优雅的界面，后端程序员也不得不介入功能实现，因为很多时候前端和后端无法完全划分，仅仅靠前端开发者是无法设计出优美而又可用的界面的。①

① 我并不是鼓励后端开发者越俎代庖，只是建议后端开发者略微了解前端的技术，以便于在大型工程中更好地合作。同时当没有前端开发者与你合作的时候，也可以设计出不至于太难看的页面。

作为后端开发者，你可能和我一样都不太擅长设计，不过没关系，我们可以利用已有的优秀设计。如果你认同 Twitter 的简洁风格，那么 Twitter Bootstrap 是最好的选择。Twitter Bootstrap 是由 Twitter 的设计师和工程师发起的开源项目，它提供了一套与 Twitter 风格一致的简洁、优雅的 Web UI，包含了完全由 HTML、CSS、JavaScript 实现的用户交互工具。不管你是资深的前端工程师，还是专业的后端开发者，你都可以轻松地使用Twitter Bootstrap 制作出优美的界面。图5-8 是Twitter Bootstrap 部件的介绍页面。

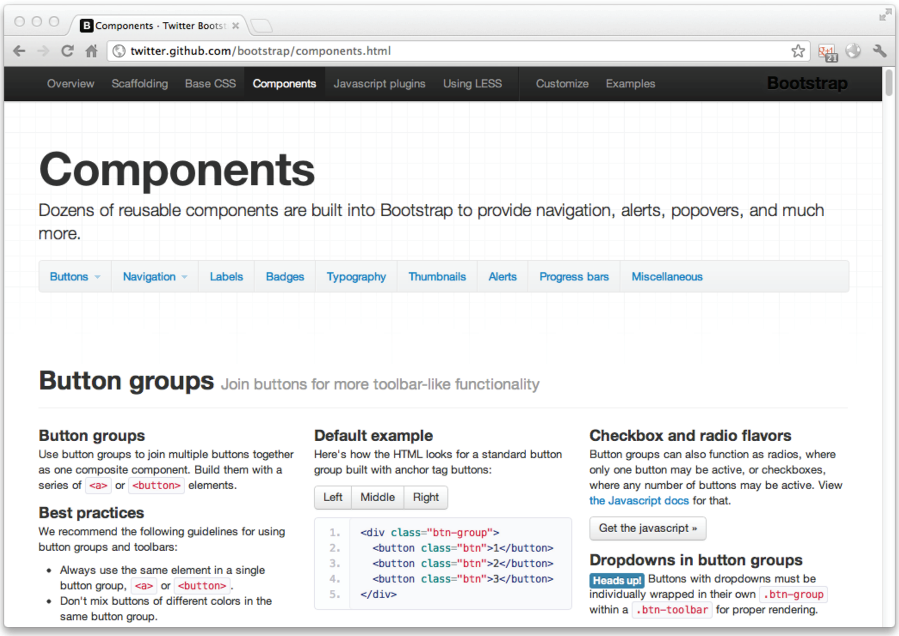


图5-8　Twitter Bootstrap

5.5.4　使用 Bootstrap

现在我们就用 Bootstrap 开始设计我们的界面。从http://twitter.github.com/bootstrap/下载bootstrap.zip，解压后可以看到以下文件：

```
css/bootstrap-responsive.css
css/bootstrap-responsive.min.css
css/bootstrap.css
css/bootstrap.min.css
img/glyphicons-halflings-white.png
img/glyphicons-halflings.png
js/bootstrap.js
js/bootstrap.min.js
```

其中所有的 JavaScript 和 CSS 文件都提供了开发版和产品版，前者是原始的代码，后者经过压缩，文件名中带有 min。将 img 目录复制到工程 public 目录下，将 bootstrap.css、bootstrap-responsive.css 复制到 public/stylesheets 中，将 bootstrap.js 复制到 public/javascripts 目录中，然后从http://jquery.com/下载一份最新版的 jquery.js 也放入 public/javascripts 目录中。

接下来，修改 views/layout.ejs：

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html>
  <head>
    <title><%= title %> - Microblog</title>
    <link rel='stylesheet' href='/stylesheets/bootstrap.css' />
    <style type="text/css">
      body {
        padding-top: 60px;
        padding-bottom: 40px;
      }
    </style>
    <link href="stylesheets/bootstrap-responsive.css" rel="stylesheet">
  </head>
  <body>

    <div class="navbar navbar-fixed-top">
      <div class="navbar-inner">
        <div class="container">
          <a class="btn btn-navbar" data-toggle="collapse" data-target=".nav-collapse">
            <span class="icon-bar"></span>
            <span class="icon-bar"></span>
            <span class="icon-bar"></span>
          </a>
          <a class="brand" href="/">Microblog</a>
          <div class="nav-collapse">
            <ul class="nav">
              <li class="active"><a href="/">首页</a></li>
              <li><a href="/login">登入</a></li>
              <li><a href="/reg">注册</a></li>
            </ul>
          </div>
        </div>
      </div>
    </div>

    <div id="container" class="container">
      <%- body %>
```

5

```
      <hr />
      <footer>
        <p><a href="http://www.byvoid.com/" target="_blank">BYVoid</a> 2012</p>
      </footer>
    </div>
  </body>
  <script src="/javascripts/jquery.js"></script>
  <script src="/javascripts/bootstrap.js"></script>
</html>
```

上面代码是使用Bootstrap部件实现的一个简单页面框架，整个页面分为顶部工具栏、正文和页脚三部分，其中正文和页脚包含在名为 container 的 div 标签中。

最后我们设计首页，修改 views/index.ejs：

```
<div class="hero-unit">
  <h1>欢迎来到 Microblog</h1>
  <p>Microblog 是一个基于 Node.js 的微博系统。</p>
  <p>
    <a class="btn btn-primary btn-large" href="/login">登录</a>
    <a class="btn btn-large" href="/reg">立即注册</a>
  </p>
</div>

<div class="row">
  <div class="span4">
    <h2>Carbo 说</h2>
     <p>东风破早梅 向暖一枝开 冰雪无人见 春从天上来</p>
  </div>
  <div class="span4">
    <h2>BYVoid 说</h2>
     <p>
     Open Chinese Convert (OpenCC) 是一个开源的中文简繁转换项目，
     致力于制作高质量的基于统计预料的简繁转换词库。
     还提供函数库(libopencc)、命令行简繁转换工具、人工校对工具、词典生成程序、
     在线转换服务及图形用户界面。</p>
 </div>
  <div class="span4">
    <h2>佛振 说</h2>
    <p>中州韵输入法引擎 / Rime Input Method Engine 取意历史上通行的中州韵，
    愿写就一部汇集音韵学智慧的输入法经典之作。
    项目网站设在 http://code.google.com/p/rimeime/
    创造应用价值是一方面，更要坚持对好技术的追求，希望能写出灵动而易于扩展的代码，
    使其成为一款个性十足的开源输入法。</p>
  </div>
</div>
```

首页的效果如图5-9 所示。



图5-9　使用 Bootstrap 实现的首页

怎么样？即使不懂设计也做出了优雅的界面，使用 Bootstrap 可以大大简化前端设计工作。

5.6　用户注册和登录

在上一节我们使用 Bootstrap 创建了网站的基本框架。在这一节我们要实现用户会话的功能，包括用户注册和登录状态的维护。为了实现这些功能，我们需要引入会话机制来记录用户状态，还要访问数据库来保存和读取用户信息。现在就让我们从数据库开始。

5.6.1　访问数据库

我们选用 MongoDB 作为网站的数据库系统，它是一个开源的 NoSQL 数据库，相比 MySQL 那样的关系型数据库，它更为轻巧、灵活，非常适合在数据规模很大、事务性不强的场合下使用。

1. NoSQL

什么是 NoSQL 呢？为了解释清楚，首先让我们来介绍几个概念。在传统的数据库中，

数据库的格式是由表（table）、行（row）、字段（field）组成的。表有固定的结构，规定了每行有哪些字段，在创建时被定义，之后修改很困难。行的格式是相同的，由若干个固定的字段组成。每个表可能有若干个字段作为索引（index），这其中有的是主键（primary key），用于约束表中的数据，还有唯一键（unique key），确保字段中不存放重复数据。表和表之间可能还有相互的约束，称为外键（foreign key）。对数据库的每次查询都要以行为单位，复杂的查询包括嵌套查询、连接查询和交叉表查询。

拥有这些功能的数据库被称为关系型数据库，关系型数据库通常使用一种叫做 SQL（Structured Query Language）的查询语言作为接口，因此又称为 SQL 数据库。典型的 SQL 数据库有 MySQL、Oracle、Microsoft SQL Server、PostgreSQL、SQLite，等等。

NoSQL 是 1998 年被提出的，它曾经是一个轻量、开源、不提供SQL功能的关系数据库。但现在 NoSQL 被认为是 Not Only SQL 的简称，主要指非关系型、分布式、不提供 ACID[①]的数据库系统。正如它的名称所暗示的，NoSQL 设计初衷并不是为了取代 SQL 数据库的，而是作为一个补充，它和 SQL 数据库有着各自不同的适应领域。NoSQL 不像 SQL 数据库一样都有着统一的架构和接口，不同的 NoSQL 数据库系统从里到外可能完全不同。

2. MongoDB

MongoDB 是一个对象数据库，它没有表、行等概念，也没有固定的模式和结构，所有的数据以文档的形式存储。所谓文档就是一个关联数组式的对象，它的内部由属性组成，一个属性对应的值可能是一个数、字符串、日期、数组，甚至是一个嵌套的文档。下面是一个 MongoDB 文档的示例：

```
{ "_id" : ObjectId( "4f7fe8432b4a1077a7c551e8" ),
  "uid" : 2004,
  "username" : "byvoid",
  "net9" : { "nickname" : "BYVoid",
    "surname" : "Kuo",
    "givenname" : "Carbo",
    "fullname" : "Carbo Kuo",
    "emails" : [ "byvoid@byvoid.com", "byvoid.kcp@gmail.com" ],
    "website" : "http://www.byvoid.com",
    "address" : "Zijing 2#, Tsinghua University" }
}
```

上面文档中 `uid` 是一个整数属性，`username` 是字符串属性，`_id` 是文档对象的标识符，格式为特定的 `ObjectId`。`net9` 是一个嵌套的文档，其内部结构与一般文档无异。从格式来看文档好像 JSON，没错，MongoDB 的数据格式就是 JSON[②]，因此与 JavaScript 的

① ACID 是数据库系统中事务（transaction）所必须具备的四个特性，即原子性（atomicity）、一致性（consistency）、隔离性（isolation）和持久性（durability）。

② 准确地说，MongoDB 的数据格式是 BSON （Binary JSON），它是 JSON 的一个扩展。

亲和性很强。在 Mongodb 中对数据的操作都是以文档为单位的，当然我们也可以修改文档的部分属性。对于查询操作，我们只需要指定文档的任何一个属性，就可在数据库中将满足条件的所有文档筛选出来。为了加快查询，MongoDB 也对文档实现了索引，这一点和 SQL 数据库一样。

3. 连接数据库

现在，让我们来看看如何连接数据库吧。首先确保已在本地安装好了 MongoDB，如果没有，请去http://www.mongodb.org/查看如何安装。

为了在 Node.js 中使用 MongoDB，我们需要获取一个模块。打开工程目录中的 package.json，在 dependencies 属性中添加一行代码：

```
{
    "name": "microblog"
  , "version": "0.0.1"
  , "private": true
  , "dependencies": {
      "express": "2.5.8"
    , "ejs": ">= 0.0.1"
    , "mongodb": ">= 0.9.9"
  }
}
```

然后运行 npm install 更新依赖的模块。接下来在工程的目录中创建 settings.js 文件，这个文件用于保存数据库的连接信息。我们将用到的数据库命名为 microblog，数据库服务器在本地，因此Settings.js文件的内容如下：

```
module.exports = {
  cookieSecret: 'microblogbyvoid',
  db: 'microblog',
  host: 'localhost',
};
```

其中，db 是数据库的名称，host 是数据库的地址。cookieSecret 用于 Cookie 加密与数据库无关，我们留作后用。

接下来在 models 子目录中创建 db.js，内容是：

```
var settings = require('../settings');
var Db = require('mongodb').Db;
var Connection = require('mongodb').Connection;
var Server = require('mongodb').Server;

module.exports = new Db(settings.db, new Server(settings.host, Connection.DEFAULT_
  PORT, {}));
```

以上代码通过 module.exports 输出了创建的数据库连接，在后面的小节中我们会用到这个模块。由于模块只会被加载一次，以后我们在其他文件中使用时均为这一个实例。

5.6.2　会话支持

在完成用户注册和登录功能之前，我们需要先了解会话的概念。会话是一种持久的网络协议，用于完成服务器和客户端之间的一些交互行为。会话是一个比连接粒度更大的概念，一次会话可能包含多次连接，每次连接都被认为是会话的一次操作。在网络应用开发中，有必要实现会话以帮助用户交互。例如网上购物的场景，用户浏览了多个页面，购买了一些物品，这些请求在多次连接中完成。许多应用层网络协议都是由会话支持的，如 FTP、Telnet 等，而 HTTP 协议是无状态的，本身不支持会话，因此在没有额外手段的帮助下，前面场景中服务器不知道用户购买了什么。

为了在无状态的 HTTP 协议之上实现会话，Cookie 诞生了。Cookie 是一些存储在客户端的信息，每次连接的时候由浏览器向服务器递交，服务器也向浏览器发起存储 Cookie 的请求，依靠这样的手段服务器可以识别客户端。我们通常意义上的 HTTP 会话功能就是这样实现的。具体来说，浏览器首次向服务器发起请求时，服务器生成一个唯一标识符并发送给客户端浏览器，浏览器将这个唯一标识符存储在 Cookie 中，以后每次再发起请求，客户端浏览器都会向服务器传送这个唯一标识符，服务器通过这个唯一标识符来识别用户。

对于开发者来说，我们无须关心浏览器端的存储，需要关注的仅仅是如何通过这个唯一标识符来识别用户。很多服务端脚本语言都有会话功能，如 PHP，把每个唯一标识符存储到文件中。Express 也提供了会话中间件，默认情况下是把用户信息存储在内存中，但我们既然已经有了 MongoDB，不妨把会话信息存储在数据库中，便于持久维护。为了使用这一功能，我们首先要获得一个叫做 connect-mongo 的模块，在 package.json 中添加一行代码：

```
{
    "name": "microblog"
  , "version": "0.0.1"
  , "private": true
  , "dependencies": {
      "express": "2.5.8"
    , "ejs": ">= 0.0.1"
    , "connect-mongo": ">= 0.1.7"
    , "mongodb": ">= 0.9.9"
  }
}
```

运行 npm install 获得模块。然后打开 app.js，添加以下内容：

```
var MongoStore = require('connect-mongo');
var settings = require('../settings');
```

```
app.configure(function(){
  app.set('views', __dirname + '/views');
  app.set('view engine', 'ejs');
  app.use(express.bodyParser());
  app.use(express.methodOverride());
  app.use(express.cookieParser());
  app.use(express.session({
    secret: settings.cookieSecret,
    store: new MongoStore({
      db: settings.db
    })
  }));
  app.use(app.router);
  app.use(express.static(__dirname + '/public'));
});
```

其中 `express.cookieParser()` 是 Cookie 解析的中间件。`express.session()` 则提供会话支持，设置它的 `store` 参数为 `MongoStore` 实例，把会话信息存储到数据库中，以避免丢失。

在后面的小节中，我们可以通过 `req.session` 获取当前用户的会话对象，以维护用户相关的信息。

5.6.3 注册和登入

我们已经准备好了数据库访问和会话存储的相关信息，接下来开始实现网站的第一个功能，用户注册和登入。

1. 注册页面

首先来设计用户注册页面的表单，创建 views/reg.ejs 文件，内容是：

```
<form class="form-horizontal" method="post">
  <fieldset>
    <legend>用户注册</legend>
    <div class="control-group">
      <label class="control-label" for="username">用户名</label>
      <div class="controls">
        <input type="text" class="input-xlarge" id="username" name="username">
        <p class="help-block">你的账户名称，用于登录和显示。</p>
      </div>
    </div>
    <div class="control-group">
      <label class="control-label" for="password">口令</label>
      <div class="controls">
        <input type="password" class="input-xlarge" id="password" name="password">
      </div>
```

```
      </div>
      <div class="control-group">
        <label class="control-label" for="password-repeat">重复输入口令</label>
        <div class="controls">
          <input type="password" class="input-xlarge" id="password-repeat"
            name="password-repeat">
        </div>
      </div>
      <div class="form-actions">
        <button type="submit" class="btn btn-primary">注册</button>
      </div>
    </fieldset>
  </form>
```

这个表单中有3个输入单元，分别是 `username`、`password` 和 `password-repeat`。表单的请求方法是 POST，将会发送到相同的路径下。

到目前为止我们所有的路由规则还都写在了 app.js 中，随着规模扩大其维护难度不断提高，因此我们需要把所有的路由规则分离出去。修改 app.js 的 `app.configure` 部分，用 `app.use(express.router(routes))` 代替 `app.use(app.router)`：

```
app.configure(function(){
  app.set('views', --dirname + '/views');
  app.set('view engine', 'ejs');
  app.use(express.bodyParser());
  app.use(express.methodOverride());
  app.use(express.cookieParser());
  app.use(express.session({
    secret: settings.cookieSecret,
    store: new MongoStore({
      db: settings.db
    })
  }));
  app.use(express.router(routes));
  app.use(express.static(--dirname + '/public'));
});
```

接下来打开 routes/index.js，把内容改为：

```
module.exports = function(app) {
  app.get('/', function(req, res) {
    res.render('index', {
      title: '首页'
    });
  });

  app.get('/reg', function(req, res) {
    res.render('reg', {
```

```
      title: '用户注册',
    });
  });
};
```

现在运行 app.js，在浏览器中打开 http://localhost:3000/reg，可以看到如图5-10所示的页面。

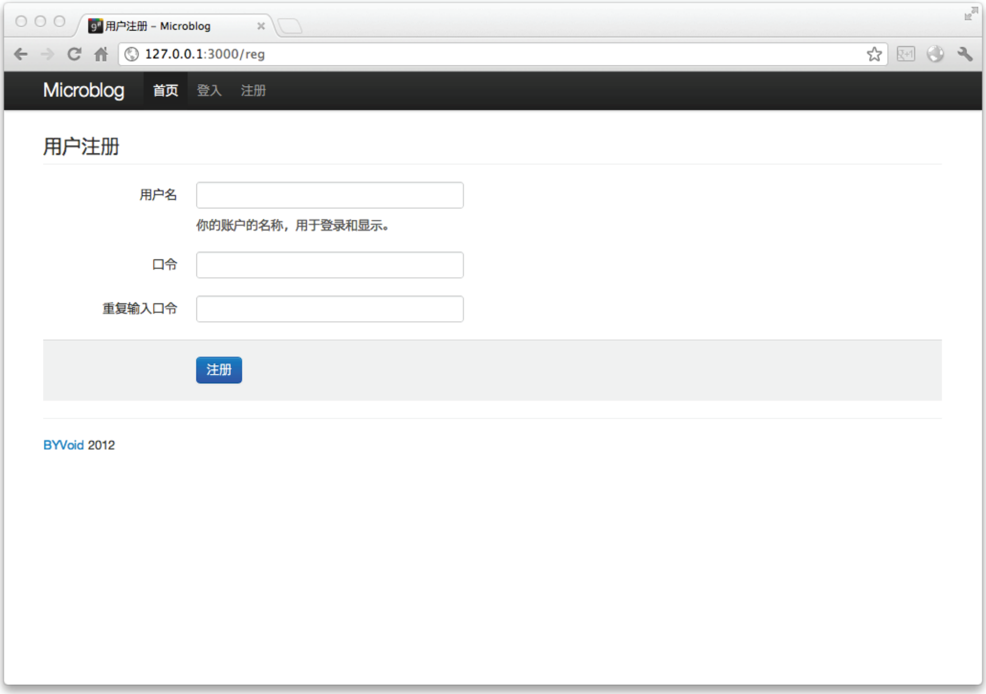


图5-10 注册页面的效果

2. 注册响应

上面这个页面十分简洁优雅，看了以后是不是有立即注册的冲动呢？当然，现在点击注册是没有效果的，因为我们还没有实现 POST 请求发送后的功能，下面就来实现。在 routes/index.js 中添加 /reg 的 POST 响应函数：

```
app.post('/reg', function(req, res) {
  //检验用户两次输入的口令是否一致
  if (req.body['password-repeat'] != req.body['password']) {
    req.flash('error', '两次输入的口令不一致');
    return res.redirect('/reg');
  }
```

5

```
  //生成口令的散列值
  var md5 = crypto.createHash('md5');
  var password = md5.update(req.body.password).digest('base64');

  var newUser = new User({
    name: req.body.username,
    password: password,
  });

  //检查用户名是否已经存在
  User.get(newUser.name, function(err, user) {
    if (user)
      err = 'Username already exists.';
    if (err) {
      req.flash('error', err);
      return res.redirect('/reg');
    }
    //如果不存在则新增用户
    newUser.save(function(err) {
      if (err) {
        req.flash('error', err);
        return res.redirect('/reg');
      }
      req.session.user = newUser;
      req.flash('success', '注册成功');
      res.redirect('/');
    });
  });
});
```

这段代码用到了一些新的东西，我们一一说明。

- `req.body` 就是 POST 请求信息解析过后的对象，例如我们要访问用户传递的 `password` 域的值，只需访问 `req.body['password']` 即可。
- `req.flash` 是 Express 提供的一个奇妙的工具，通过它保存的变量只会在用户当前和下一次的请求中被访问，之后会被清除，通过它我们可以很方便地实现页面的通知和错误信息显示功能。
- `res.redirect` 是重定向功能，通过它会向用户返回一个 303 See Other 状态，通知浏览器转向相应页面。
- `crypto` 是 Node.js 的一个核心模块，功能是加密并生成各种散列，使用它之前首先要声明 `var crypto = require('crypto')`。我们代码中使用它计算了密码的散列值。
- `User` 是我们设计的用户对象，在后面我们会详细介绍，这里先假设它的接口都是可用的，使用前需要通过 `var User = require('../models/user.js')` 引用。

- User.get 的功能是通过用户名获取已知用户，在这里我们判断用户名是否已经存在。User.save 可以将用户对象的修改写入数据库。
- 通过 req.session.user = newUser 向会话对象写入了当前用户的信息，在后面我们会通过它判断用户是否已经登录。

3. 用户模型

在前面的代码中，我们直接使用了 User 对象。User 是一个描述数据的对象，即 MVC 架构中的模型。前面我们使用了许多视图和控制器，这是第一次接触到模型。与视图和控制器不同，模型是真正与数据打交道的工具，没有模型，网站就只是一个外壳，不能发挥真实的作用，因此它是框架中最根本的部分。现在就让我们来实现 User 模型吧。

在 models 目录中创建 user.js 的文件，内容如下：

```
var mongodb = require('./db');

function User(user) {
  this.name = user.name;
  this.password = user.password;
};
module.exports = User;

User.prototype.save = function save(callback) {
  // 存入 Mongodb 的文档
  var user = {
    name: this.name,
    password: this.password,
  };
  mongodb.open(function(err, db) {
    if (err) {
      return callback(err);
    }
    // 读取 users 集合
    db.collection('users', function(err, collection) {
      if (err) {
        mongodb.close();
        return callback(err);
      }
      // 为 name 属性添加索引
      collection.ensureIndex('name', {unique: true});
      // 写入 user 文档
      collection.insert(user, {safe: true}, function(err, user) {
        mongodb.close();
        callback(err, user);
      });
    });
```

```
  });
};

User.get = function get(username, callback) {
  mongodb.open(function(err, db) {
    if (err) {
      return callback(err);
    }
    // 读取 users 集合
    db.collection('users', function(err, collection) {
      if (err) {
        mongodb.close();
        return callback(err);
      }
      // 查找 name 属性为 username 的文档
      collection.findOne({name: username}, function(err, doc) {
        mongodb.close();
        if (doc) {
          // 封装文档为 User 对象
          var user = new User(doc);
          callback(err, user);
        } else {
          callback(err, null);
        }
      });
    });
  });
};
```

以上代码实现了两个接口，`User.prototype.save` 和 `User.get`，前者是对象实例的方法，用于将用户对象的数据保存到数据库中，后者是对象构造函数的方法，用于从数据库中查找指定的用户。

4. 视图交互

现在几乎已经万事俱备，只差视图的支持了。为了实现不同登录状态下页面呈现不同内容的功能，我们需要创建动态视图助手，通过它我们才能在视图中访问会话中的用户数据。同时为了显示错误和成功的信息，也要在动态视图助手中增加响应的函数。

打开 app.js，添加以下代码：

```
app.dynamicHelpers({
  user: function(req, res) {
    return req.session.user;
  },
  error: function(req, res) {
    var err = req.flash('error');
```

```
    if (err.length)
      return err;
    else
      return null;
  },
  success: function(req, res) {
    var succ = req.flash('success');
    if (succ.length)
      return succ;
    else
      return null;
  },
});
```

接下来，修改 layout.ejs中的导航栏部分：

```
<ul class="nav">
  <li class="active"><a href="/">首页</a></li>
  <% if (!user) { %>
    <li><a href="/login">登入</a></li>
    <li><a href="/reg">注册</a></li>
  <% } else { %>
    <li><a href="/logout">登出</a></li>
  <% } %>
</ul>
```

上面功能是为已登入用户和未登入用户显示不同的信息。在 container 中，<%- body %>之前加入：

```
<% if (success) { %>
  <div class="alert alert-success">
    <%= success %>
  </div>
<% } %>
<% if (error) { %>
  <div class="alert alert-error">
    <%= error %>
  </div>
<% } %>
```

它的功能是页面通知。

现在看看最终的效果吧，图5-11和图5-12分别是注册时遇到错误和注册成功以后的画面。

5

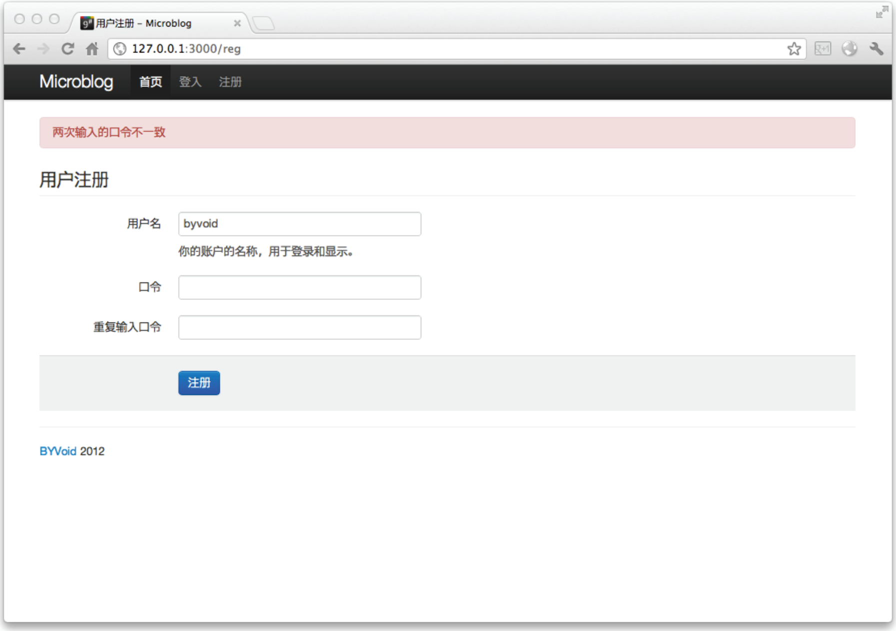


图5-11　两次输入的密码不一致



图5-12　注册成功

5. 登入和登出

当我们完成用户注册的功能以后，再实现用户登入和登出就相当容易了。把下面的代码加到 routes/index.js 中：

```
app.get('/login', function(req, res) {
  res.render('login', {
    title: '用户登入',
  });
});

app.post('/login', function(req, res) {
  //生成口令的散列值
  var md5 = crypto.createHash('md5');
  var password = md5.update(req.body.password).digest('base64');

  User.get(req.body.username, function(err, user) {
    if (!user) {
      req.flash('error', '用户不存在');
      return res.redirect('/login');
    }
    if (user.password != password) {
      req.flash('error', '用户口令错误');
      return res.redirect('/login');
    }
    req.session.user = user;
    req.flash('success', '登入成功');
    res.redirect('/');
  });
});

app.get('/logout', function(req, res) {
  req.session.user = null;
  req.flash('success', '登出成功');
  res.redirect('/');
});
```

在这里你可以清晰地看出登入和登出仅仅是 `req.session.user` 变量的标记，非常简单。但这会不会有安全性问题呢？不会的，因为这个变量只有服务端才能访问到，只要不是黑客攻破了整个服务器，无法从外部改动。

最后我们创建 views/login.ejs，内容如下：

```
<form class="form-horizontal" method="post">
  <fieldset>
    <legend>用户登入</legend>
    <div class="control-group">
      <label class="control-label" for="username">用户名</label>
```

5

```
        <div class="controls">
          <input type="text" class="input-xlarge" id="username" name="username">
        </div>
      </div>
      <div class="control-group">
        <label class="control-label" for="password">口令</label>
        <div class="controls">
          <input type="password" class="input-xlarge" id="password" name="password">
        </div>
      </div>
      <div class="form-actions">
        <button type="submit" class="btn btn-primary">登入</button>
      </div>
    </fieldset>
  </form>
```

在浏览器中访问http://localhost:3000/login，你将会看到如图5-13 所示的页面。

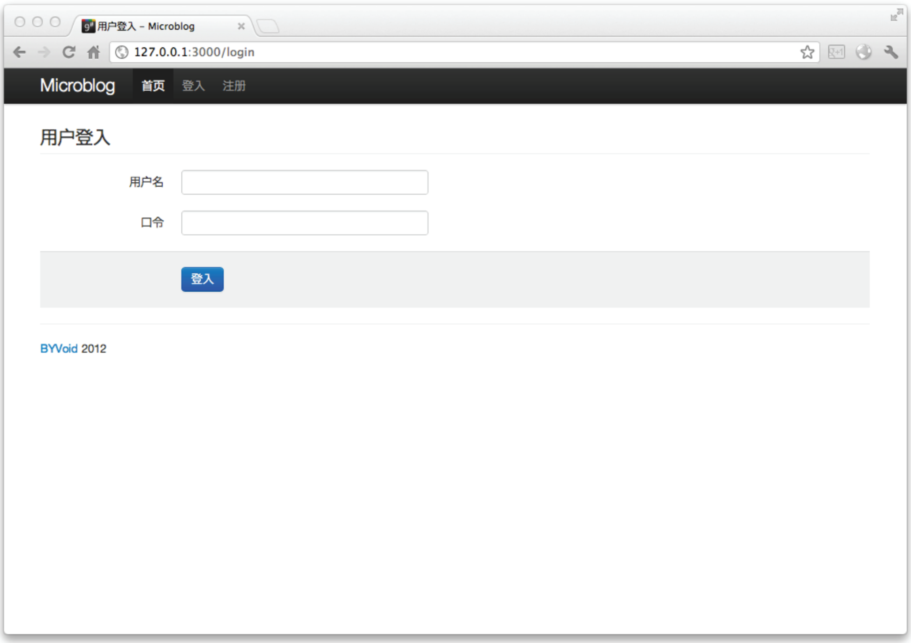


图5-13　用户登入

至此用户注册和登录的功能就完全实现了。

5.6.4　页面权限控制

在前面我们已经实现了用户登入，并且在页面中通过不同的内容反映出了用户已登入和

未登入的状态。现在我们还有一个工作要做，就是为页面设置访问权限。例如，登出功能应该只对已登入的用户开放，注册和登入页面则应该阻止已登入的用户访问。如何实现这一点呢？最简单的方法是在每个页面的路由响应函数内检查用户是否已经登录，但这会带来很多重复的代码，违反了 DRY [①]原则。因此，我们利用路由中间件来实现这个功能。

5.3.5 节介绍了同一路径绑定多个响应函数的方法，通过调用 `next()` 转移控制权，这种方法叫做**路由中间件**。我们可以把用户登入状态检查放到路由中间件中，在每个路径前增加路由中间件，即可实现页面权限控制。

最终的 routes/index.js 内容如下：

```
var crypto = require('crypto');
var User = require('../models/user.js');

module.exports = function(app) {
  app.get('/', function(req, res) {
    res.render('index', {
      title: '首页'
    });
  });

  app.get('/reg', checkNotLogin);
  app.get('/reg', function(req, res) {
    res.render('reg', {
      title: '用户注册',
    });
  });

  app.post('/reg', checkNotLogin);
  app.post('/reg', function(req, res) {
    //检验用户两次输入的口令是否一致
    if (req.body['password-repeat'] != req.body['password']) {
      req.flash('error', '两次输入的口令不一致');
      return res.redirect('/reg');
    }

    //生成口令的散列值
    var md5 = crypto.createHash('md5');
    var password = md5.update(req.body.password).digest('base64');

    var newUser = new User({
      name: req.body.username,
      password: password,
    });
```

① DRY（Don't Repeat Yourself）是软件工程设计的一个基本原则，又称“一次且仅一次”（Once And Only Once），指的是开发中应该避免相同意义的代码重复出现。

5

```
    //检查用户名是否已经存在
    User.get(newUser.name, function(err, user) {
      if (user)
        err = 'Username already exists.';
      if (err) {
        req.flash('error', err);
        return res.redirect('/reg');
      }
      //如果不存在则新增用户
      newUser.save(function(err) {
        if (err) {
          req.flash('error', err);
          return res.redirect('/reg');
        }
        req.session.user = newUser;
        req.flash('success', '注册成功');
        res.redirect('/');
      });
    });
  });

  app.get('/login', checkNotLogin);
  app.get('/login', function(req, res) {
    res.render('login', {
      title: '用户登入',
    });
  });

  app.post('/login', checkNotLogin);
  app.post('/login', function(req, res) {
    //生成口令的散列值
    var md5 = crypto.createHash('md5');
    var password = md5.update(req.body.password).digest('base64');

    User.get(req.body.username, function(err, user) {
      if (!user) {
        req.flash('error', '用户不存在');
        return res.redirect('/login');
      }
      if (user.password != password) {
        req.flash('error', '用户口令错误');
        return res.redirect('/login');
      }
      req.session.user = user;
      req.flash('success', '登入成功');
      res.redirect('/');
    });
  });
```

```
  app.get('/logout', checkLogin);
  app.get('/logout', function(req, res) {
    req.session.user = null;
    req.flash('success', '登出成功');
    res.redirect('/');
  });
};

function checkLogin(req, res, next) {
  if (!req.session.user) {
    req.flash('error', '未登入');
    return res.redirect('/login');
  }
  next();
}

function checkNotLogin(req, res, next) {
  if (req.session.user) {
    req.flash('error', '已登入');
    return res.redirect('/');
  }
  next();
}
```


5.7　发表微博

现在网站已经具备了用户注册、登入、页面权限控制的功能，这些功能为网站最核心的部分——发表微博做好了准备。在这个小节里，我们将会实现发表微博的功能，完成整个网站的设计。

5.7.1　微博模型

现在让我们从模型开始设计。仿照用户模型，将微博模型命名为 `Post` 对象，它拥有与 `User` 相似的接口，分别是 `Post.get` 和 `Post.prototype.save`。`Post.get` 的功能是从数据库中获取微博，可以按指定用户获取，也可以获取全部的内容。`Post.prototype.save` 是 `Post` 对象实例的方法，用于将对象的变动保存到数据库。

创建 models/post.js，写入以下内容：

```
var mongodb = require('./db');

function Post(username, post, time) {
  this.user = username;
  this.post = post;
```

```
  if (time) {
    this.time = time;
  } else {
    this.time = new Date();
  }
};
module.exports = Post;

Post.prototype.save = function save(callback) {
  // 存入 Mongodb 的文档
  var post = {
    user: this.user,
    post: this.post,
    time: this.time,
  };
  mongodb.open(function(err, db) {
    if (err) {
      return callback(err);
    }
    // 读取 posts 集合
    db.collection('posts', function(err, collection) {
      if (err) {
        mongodb.close();
        return callback(err);
      }
      // 为 user 属性添加索引
      collection.ensureIndex('user');
      // 写入 post 文档
      collection.insert(post, {safe: true}, function(err, post) {
        mongodb.close();
        callback(err, post);
      });
    });
  });
};

Post.get = function get(username, callback) {
  mongodb.open(function(err, db) {
    if (err) {
      return callback(err);
    }
    // 读取 posts 集合
    db.collection('posts', function(err, collection) {
      if (err) {
        mongodb.close();
        return callback(err);
      }
```

```
      // 查找 user 属性为 username 的文档，如果 username 是 null 则匹配全部
      var query = {};
      if (username) {
        query.user = username;
      }
      collection.find(query).sort({time: -1}).toArray(function(err, docs) {
        mongodb.close();
        if (err) {
          callback(err, null);
        }
        // 封装 posts 为 Post 对象
        var posts = [];
        docs.forEach(function(doc, index) {
          var post = new Post(doc.user, doc.post, doc.time);
          posts.push(post);
        });
        callback(null, posts);
      });
    });
  });
};
```


在后面我们会通过控制器调用这个模块。

5.7.2 发表微博

我们曾经约定通过 POST 方法访问 `/post` 以发表微博，现在让我们来实现这个控制器。在 routes/index.js 中添加下面的代码：

```
app.post('/post', checkLogin);
app.post('/post', function(req, res) {
  var currentUser = req.session.user;
  var post = new Post(currentUser.name, req.body.post);
  post.save(function(err) {
    if (err) {
      req.flash('error', err);
      return res.redirect('/');
    }
    req.flash('success', '发表成功');
    res.redirect('/u/' + currentUser.name);
  });
});
```

这段代码通过 `req.session.user` 获取当前用户信息，从 `req.body.post` 获取用户发表的内容，建立 `Post` 对象，调用 `save()` 方法存储信息，最后将用户重定向到用户页面。

5.7.3　用户页面

用户页面的功能是展示用户发表的所有内容，在routes/index.js中加入以下代码：

```
app.get('/u/:user', function(req, res) {
  User.get(req.params.user, function(err, user) {
    if (!user) {
      req.flash('error', '用户不存在');
      return res.redirect('/');
    }
    Post.get(user.name, function(err, posts) {
      if (err) {
        req.flash('error', err);
        return res.redirect('/');
      }
      res.render('user', {
        title: user.name,
        posts: posts,
      });
    });
  });
});
```

它的功能是首先检查用户是否存在，如果存在则从数据库中获取该用户的微博，最后通过 posts 属性传递给 user 视图。views/user.ejs 的内容如下：

```
<% if (user) { %>
  <%- partial('say') %>
<% } %>
<%- partial('posts') %>
```

根据 DRY 原则，我们把重复用到的部分都提取出来，分别放入 say.ejs 和 posts.ejs。say.ejs 的功能是显示一个发表微博的表单，它的内容如下：

```
<form method="post" action="/post" class="well form-inline center" style="text-align:
     center;">
  <input type="text" class="span8" name="post">
  <button type="submit" class="btn btn-success"><i class="icon-comment icon-white">
      </i> 发言</button>
</form>
```

posts.ejs 的目的是按照行列显示传入的 posts 的所有内容：

```
<% posts.forEach(function(post, index) {
  if (index % 3 == 0) { %>
    <div class="row">
<%} %>
  <div class="span4">
```

```
    <h2><a href="/u/<%= post.user %>"><%= post.user %></a> 说</h2>
    <p><small><%= post.time %></small></p>
    <p><%= post.post %></p>
  </div>
<% if (index % 3 == 2) { %>
    </div><!-- end row -->
<% } %>
<%}) %>
<% if (posts.length % 3 != 0) { %>
  </div><!-- end row -->
<%} %>
```

完成上述工作后，重启服务器。在用户的页面上发表几个微博，然后可以看到用户页面的效果如图5-14 所示。

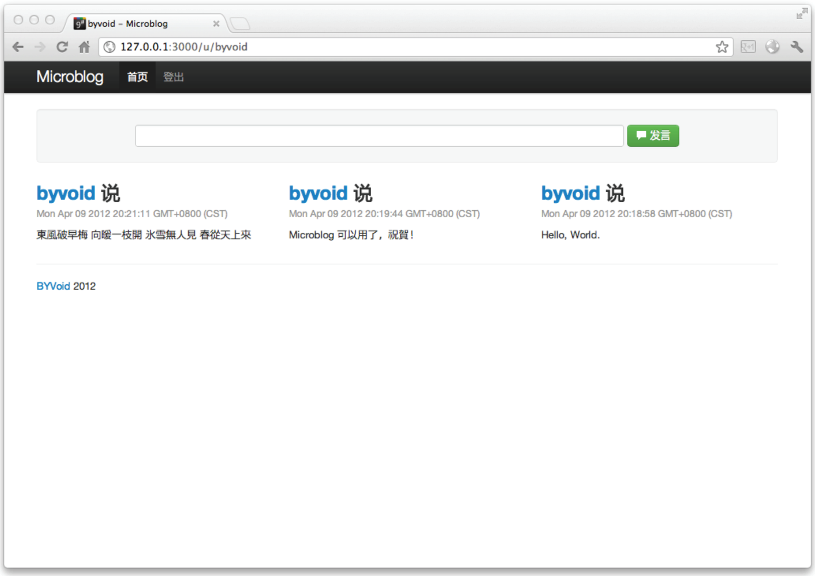


图5-14 用户页面

5.7.4 首页

最后一步是实现首页的内容。我们计划在首页显示所有用户发表的微博，按时间从新到旧的顺序。

在 routes/index.js 中添加下面代码：

```
app.get('/', function(req, res) {
  Post.get(null, function(err, posts) {
    if (err) {
```

```
        posts = [];
      }
      res.render('index', {
        title: '首页',
        posts: posts,
      });
    });
  });
```

它的功能是读取所有用户的微博，传递给页面 `posts` 属性。接下来修改首页的模板 index.ejs：

```
<% if (!user) { %>
  <div class="hero-unit">
    <h1>欢迎来到 Microblog</h1>
    <p>Microblog 是一个基于 Node.js 的微博系统。</p>
    <p>
      <a class="btn btn-primary btn-large" href="/login">登录</a>
      <a class="btn btn-large" href="/reg">立即注册</a>
    </p>
  </div>
<% } else { %>
  <%- partial('say') %>
<% } %>
<%- partial('posts') %>
```

下面看看首页的效果吧，图5-15和图5-16是用户登入之前和登入以后看到的首页效果。

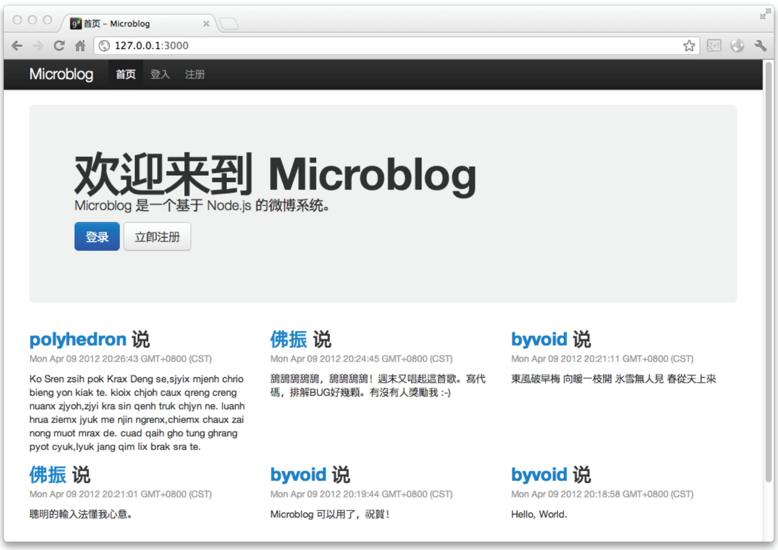


图5-15　登入之前的首页

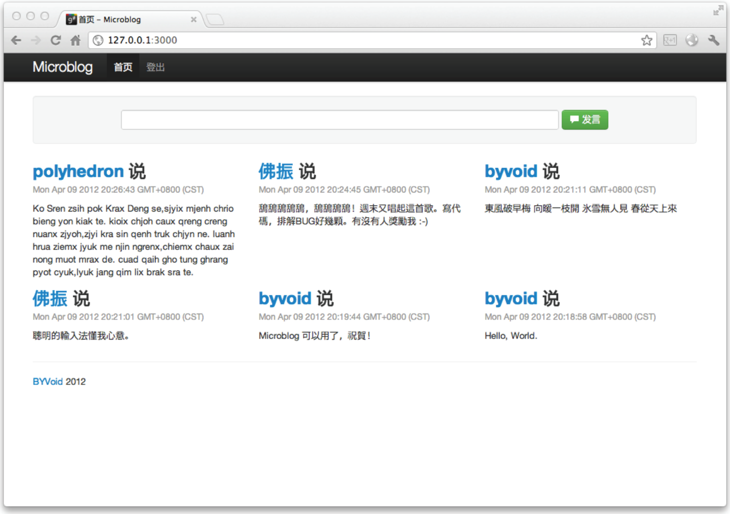


图5-16　登入以后的首页

5.7.5　下一步

到此为止，微博网站的基本功能就完成了。这个网站仅仅是微博的一个雏形，距离真正的微博还有很大的距离。例如，我们没有对注册信息进行完整的验证，如用户名的规则，密码的长短等。为了防止恶意注册还应该带有验证码和邮件认证的功能，甚至还应该支持OAuth。我们对发帖没有进行任何限制，尽管注入 HTML 是不可能的，但至少还应该对长度有限制。首页和用户页面的显示都是没有数量限制的，当微博很多以后这个页面可能会很长，应该实现分页的功能。作为社交工具，最重要的用户关注、转帖、评论、圈点用户这些功能都没有实现。

除了功能上的不足，这个网站还有潜在的性能问题，例如每次查询数据库都没有限制取得的数量，还应该对一些访问频繁的页面增加缓存机制。另外，我们一直是以开发模式在运行着这个网站，没有讨论如何把它真正部署起来，我们会在下一章详细讨论。

如果你对这个用 Node.js 实现的微博网站有兴趣，请访问https://github.com/BYVoid/microblog，这里有 Microblog 示例中的完整代码，而且在其基础上还做了进一步的改进，也欢迎你为它“添砖加瓦”。

5.8　参考资料

- “Node.js简单介绍并实现一个简单的Web MVC框架”: http://club.cnodejs.org/topic/4f16442ccae1f4aa27001135。

- "A HTTP Proxy Server in 20 Lines of node.js Code": http://www.catonmat.net/http-proxy-in-nodejs/。
- "Node.js Recommended Third-party Modules": http://nodejs.org/api/appendix_1.html。
- Express Guide: http://expressjs.com/guide.html。
- EJS：Embedded JavaScript: http://embeddedjs.com/。
- Jade: http://jade-lang.com/。
- JSON-P: http://www.json-p.org/。
- Connect: http://www.senchalabs.org/connect/。
- "深入浅出REST": http://www.infoq.com/cn/articles/rest-introduction。
- "HTTP Verbs: 谈POST、PUT 和 PATCH 的应用": http://ihower.tw/blog/archives/6483。
- Template engine (Web): http://en.wikipedia.org/wiki/Template_engine_(Web)。
- Bootstrap: http://twitter.github.com/bootstrap/。
- MongoDB Manual: http://www.mongodb.org/display/DOCS/Manual。

Node.js进阶话题

第 6 章

在本书的最后一章，我们打算讨论几个独立的话题，主要内容包括：

- 模块加载机制；
- 异步编程模式下的控制流；
- Node.js 应用部署；
- Node.js的一些劣势。

6.1　模块加载机制

Node.js 的模块加载对用户来说十分简单，只需调用 `require` 即可，但其内部机制较为复杂。我们通过这一节简要介绍一下 Node.js 模块加载的一些细节，帮你减少开发中可能遇到的问题。

6.1.1　模块的类型

Node.js 的模块可以分为两大类，一类是核心模块，另一类是文件模块。核心模块就是 Node.js 标准 API 中提供的模块，如 `fs`、`http`、`net`、`vm` 等，这些都是由 Node.js 官方提供的模块，编译成了二进制代码。我们可以直接通过 `require` 获取核心模块，例如 `require('fs')`。核心模块拥有最高的加载优先级，换言之如果有模块与其命名冲突，Node.js 总是会加载核心模块。

文件模块则是存储为单独的文件（或文件夹）的模块，可能是 JavaScript 代码、JSON 或编译好的 C/C++ 代码。文件模块的加载方法相对复杂，但十分灵活，尤其是和 npm 结合使用时。在不显式指定文件模块扩展名的时候，Node.js 会分别试图加上 .js、.json 和 .node扩展名。.js 是 JavaScript 代码，.json 是 JSON 格式的文本，.node 是编译好的 C/C++ 代码。

表 6-1总结了 Node.js 模块的类型，从上到下加载优先级由高到低。

表6-1　Node.js 模块的类别和加载顺序

核心模块		内　　建
文件模块	JavaScript	.js
	JSON	.json
	C/C++扩展	.node

6.1.2　按路径加载模块

文件模块的加载有两种方式，一种是按路径加载，一种是查找 node_modules 文件夹。

如果 `require` 参数以“`/`”开头，那么就以绝对路径的方式查找模块名称，例如 `require('/home/byvoid/module')` 将会按照优先级依次尝试加载 /home/byvoid/module.js 、

/home/byvoid/module.json 和 /home/byvoid/module.node。

如果 `require` 参数以“./”或“../”开头，那么则以相对路径的方式来查找模块，这种方式在应用中是最常见的。例如前面的例子中我们用了`require('./hello')`来加载同一文件夹下的hello.js。

6.1.3 通过查找 node_modules 目录加载模块

如果`require`参数不以“/”、“./”或“../”开头，而该模块又不是核心模块，那么就要通过查找 node_modules 加载模块了。我们使用npm获取的包通常就是以这种方式加载的。

在某个目录下执行命令`npm install express`，你会发现出现了一个叫做node_modules的目录，里面的结构大概如图 6-1 所示。

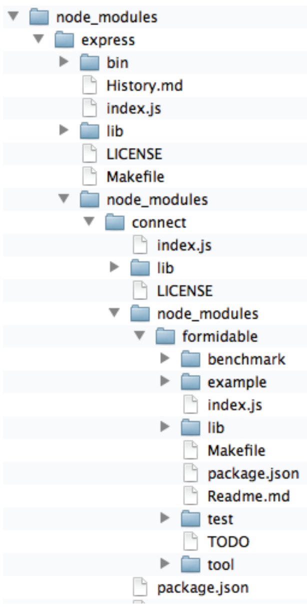


图6-1 node_modules 目录结构

在 node_modules 目录的外面一层，我们可以直接使用 `require('express')` 来代替 `require('./node_modules/express')`。这是Node.js模块加载的一个重要特性：通过查找 node_modules 目录来加载模块。

当 `require` 遇到一个既不是核心模块，又不是以路径形式表示的模块名称时，会试图在当前目录下的 node_modules 目录中来查找是不是有这样一个模块。如果没有找到，则会在当前目录的上一层中的 node_modules 目录中继续查找，反复执行这一过程，直到遇到根目录为止。举个例子，我们要在 /home/byvoid/develop/foo.js 中使用 `require('bar.js')` 命

令，Node.js会依次查找：

- /home/byvoid/develop/node_modules/bar.js
- /home/byvoid/node_modules/bar.js
- /home/node_modules/bar.js
- /node_modules/bar.js

为什么要这样做呢？因为通常一个工程内会有一些子目录，当子目录内的文件需要访问到工程共同依赖的模块时，就需要向父目录上溯了。比如说工程的目录结构如下：

```
|- project
   |- app.js
   |- models
      |- ...
   |- views
      |- ...
   |- controllers
      |- index_controller.js
      |- error_controller.js
      |- ...
   |- node_modules
      |- express
```

我们不仅要在 project 目录下的 app.js 中使用 `require('express')`，而且可能要在 controllers 子目录下的 index_controller.js 中也使用 `require('express')`，这时就需要向父目录上溯一层才能找到 node_modules 中的 express 了。

6.1.4　加载缓存

我们在前面提到过，Node.js 模块不会被重复加载，这是因为 Node.js 通过文件名缓存所有加载过的文件模块，所以以后再访问到时就不会重新加载了。注意，Node.js 是根据实际文件名缓存的，而不是 `require()` 提供的参数缓存的，也就是说即使你分别通过 `require('express')` 和 `require('./node_modules/express')` 加载两次，也不会重复加载，因为尽管两次参数不同，解析到的文件却是同一个。

6.1.5　加载顺序

下面总结一下使用 `require(some_module)` 时的加载顺序。

(1) 如果 *some_module* 是一个核心模块，直接加载，结束。

(2) 如果 *some_module* 以“/”、“./”或“../”开头，按路径加载 *some_module*，结束。

(3) 假设当前目录为 current_dir，按路径加载 current_dir/node_modules/some_module。

- 如果加载成功，结束。

- 如果加载失败，令current_dir为其父目录。
- 重复这一过程，直到遇到根目录，抛出异常，结束。

6.2　控制流

基于异步 I/O 的事件式编程容易将程序的逻辑拆得七零八落，给控制流的疏理制造障碍。让我们通过下面的例子来说明这个问题。

6.2.1　循环的陷阱

Node.js 的异步机制由事件和回调函数实现，一开始接触可能会感觉违反常规，但习惯以后就会发现还是很简单的。然而这之中其实暗藏了不少陷阱，一个很容易遇到的问题就是循环中的回调函数，初学者经常容易陷入这个圈套。让我们从一个例子开始说明这个问题。

```
//forloop.js

var fs = require('fs');
var files = ['a.txt', 'b.txt', 'c.txt'];

for (var i = 0; i < files.length; i++) {
  fs.readFile(files[i], 'utf-8', function(err, contents) {
    console.log(files[i] + ': ' + contents);
  });
}
```


这段代码的功能很直观，就是依次读取文件 a.txt、b.txt、c.txt，并输出文件名和内容。假设这三个文件的内容分别是 AAA、BBB 和 CCC，那么我们期望的输出结果就是：

```
a.txt: AAA
b.txt: BBB
c.txt: CCC
```

可是我们运行这段代码的结果是怎样的呢？竟然是这样的结果：

```
undefined: AAA
undefined: BBB
undefined: CCC
```

这个结果说明文件内容正确输出了，而文件名却不对，也就意味着，`contents` 的结果是正确的，但 `files[i]` 的值是 `undefined`。这怎么可能呢，文件名不正确却能读取文件内容？既然难以直观地理解，我们就把 `files[i]` 分解并打印出来看看，在读取文件的回调函数中分别输出 `files`、`i` 和 `files[i]`。

```
//forloopi.js

var fs = require('fs');
var files = ['a.txt', 'b.txt', 'c.txt'];

for (var i = 0; i < files.length; i++) {
  fs.readFile(files[i], 'utf-8', function(err, contents) {
    console.log(files);
    console.log(i);
    console.log(files[i]);
  });
}
```

运行修改后的代码，结果如下：

```
[ 'a.txt', 'b.txt', 'c.txt' ]
3
undefined
[ 'a.txt', 'b.txt', 'c.txt' ]
3
undefined
[ 'a.txt', 'b.txt', 'c.txt' ]
3
undefined
```

看到这里是不是有点启发了呢？三次输出的 `i` 的值都是 3，超出了 `files` 数组的下标范围，因此 `files[i]` 的值就是 `undefined` 了。这种情况通常会在 `for` 循环结束时发生，例如 `for (var i = 0; i < files.length; i++)`，退出循环时 `i` 的值就是 `files.length` 的值。既然 `i` 的值是 3，那么说明了事实上 `fs.readFile` 的回调函数中访问到的 `i` 值都是循环退出以后的，因此不能分辨。而 `files[i]` 作为 `fs.readFile` 的第一个参数在循环中就传递了，所以文件可以被定位到，而且可以显示出文件的内容。

现在问题就明朗了：原因是3次读取文件的回调函数事实上是同一个实例，其中引用到的 `i` 值是上面循环执行结束后的值，因此不能分辨。如何解决这个问题呢？我们可以利用 JavaScript 函数式编程的特性，手动建立一个闭包：

```
//forloopclosure.js

var fs = require('fs');
var files = ['a.txt', 'b.txt', 'c.txt'];

for (var i = 0; i < files.length; i++) {
  (function(i) {
    fs.readFile(files[i], 'utf-8', function(err, contents) {
      console.log(files[i] + ': ' + contents);
    });
  })(i);
}
```

上面代码在 for 循环体中建立了一个匿名函数，将循环迭代变量 i 作为函数的参数传递并调用。由于运行时闭包的存在，该匿名函数中定义的变量（包括参数表）在它内部的函数（ fs.readFile 的回调函数）执行完毕之前都不会释放，因此我们在其中访问到的 i 就分别是不同的闭包实例，这个实例是在循环体执行的过程中创建的，保留了不同的值。

事实上以上这种写法并不常见，因为它降低了程序的可读性，故不推荐使用。大多数情况下我们可以用数组的 forEach 方法解决这个问题：

```
//callbackforeach.js

var fs = require('fs');
var files = ['a.txt', 'b.txt', 'c.txt'];

files.forEach(function(filename) {
  fs.readFile(filename, 'utf-8', function(err, contents) {
    console.log(filename + ': ' + contents);
  });
});
```

6.2.2 解决控制流难题

除了循环的陷阱，Node.js 异步式编程还有一个显著的问题，即深层的回调函数嵌套。在这种情况下，我们很难像看基本控制流结构一样一眼看清回调函数之间的关系，因此当程序规模扩大时必须采取手段降低耦合度，以实现更加优美、可读的代码。这个问题本身没有立竿见影的解决方法，只能通过改变设计模式，时刻注意降低逻辑之间的耦合关系来解决。

6

除此之外，还有许多项目试图解决这一难题。async 是一个控制流解耦模块，它提供了 async.series、async.parallel、async.waterfall 等函数，在实现复杂的逻辑时使用这些函数代替回调函数嵌套可以让程序变得更清晰可读且易于维护，但你必须遵循它的编程风格。

streamlinejs和jscex则采用了更高级的手段，它的思想是“变同步为异步”，实现了一个JavaScript 到JavaScript 的编译器，使用户可以用同步编程的模式写代码，编译后执行时却是异步的。

eventproxy 的思路与前面两者区别更大，它实现了对事件发射器的深度封装，采用一种完全基于事件松散耦合的方式来实现控制流的梳理。

无论是以上哪种解决手段，都不是“非侵入性的”，也就是说它对你编程模式的影响是非常大的，你几乎不可能无代价地在使用了一种模式很久以后从容地换成另一种模式，或者直接糅合使用两种模式。而且它们都是在解决了深层嵌套的回调函数可读性问题的同时，引入了其他复杂的语法，带来了另一种可读性的降低。所以，是否使用，使用哪种方案，在决定之前是需要仔细斟酌研究的。

这些库的具体使用方法，乃至实现的原理以及更深一步的讨论已经超出了本书的范围，欢迎有兴趣的读者到Node.js中文社区（http://club.cnodejs.org/）交流。

6.3　Node.js 应用部署

在第5章我们已经使用Express实现了一个微博网站，在开发的过程中，通过`node app.js`命令运行服务器即可。但它不适合在产品环境下使用，为什么呢？因为到目前为止这个服务器还有几个重大缺陷。

- **不支持故障恢复**

  不知你是否在调试的过程中注意，当程序有错误发生时，整个进程就会结束，需要重新在终端中启动服务器。这一点在开发中无可厚非，但在产品环境下就是严重的问题了，因为一旦用户访问时触发了程序中某个隐含的bug，整个服务器就崩溃了，将无法继续为所有用户提供服务。在部署Node.js 应用的时候一定要考虑到故障恢复，提高系统的可靠性。
- **没有日志**

  对于开发者来说，日志，尤其是错误日志是及其重要的，经常查看它可以发现测试时没有注意到的程序错误。然而这个服务器运行时没有产生任何日志，包括访问日志和错误日志，所以有必要实现它的日志功能。
- **无法利用多核提高性能**

  由于Node.js是单线程的，一个进程只能利用一个CPU 核心。当请求大量到来时，单线程就成为了提高吞吐量的瓶颈。随着多核乃至众核时代的到来，只能利用一个核心所带来的浪费是十分严重的，我们需要使用多进程来提高系统的性能。
- **独占端口**

  假如整个服务器只有一个网站，或者可以给每个网站分配一个独立的IP地址，不会有端口冲突的问题。而很多时候为了充分利用服务器的资源，我们会在同一个服务器上建立多个网站，而且这些网站可能有的是PHP，有的是Rails，有的是Node.js。不能每个进程都独占80端口，所以我们有必要通过反向代理来实现基于域名的端口共享。
- **需要手动启动**

  先前每次启动服务器都是通过在命令行中直接键入命令来实现的，但在产品环境中，特别是在服务器重启以后，全部靠手动启动是不现实的。因此，我们应该制作一个自动启动服务器的脚本，并且通过该脚本可以实现停止服务器等功能。

6.3.1　日志功能

下面我们开始在第5章的代码的基础上，介绍如何实现服务器的日志功能。Express支持

两种运行模式：开发模式和产品模式，前者的目的是利于调试，后者则是利于部署。使用产品模式运行服务器的方式很简单，只需设置NODE_ENV 环境变量。通过NODE_ENV=production node app.js命令运行服务器可以看到：

```
Express server listening on port 3000 in production mode
```

接下来让我们实现访问日志和错误日志功能。访问日志就是记录用户对服务器的每个请求，包括客户端IP 地址，访问时间，访问路径，服务器响应以及客户端代理字符串。而错误日志则记录程序发生错误时的信息，由于调试中需要即时查看错误信息，将所有错误直接显示到终端即可，而在产品模式中，需要写入错误日志文件。

Express 提供了一个访问日志中间件，只需指定stream 参数为一个输出流即可将访问日志写入文件。打开app.js，在最上方加入以下代码：

```
var fs = require('fs');
var accessLogfile = fs.createWriteStream('access.log', {flags: 'a'});
var errorLogfile = fs.createWriteStream('error.log', {flags: 'a'});
```

然后在app.configure 函数第一行加入：

```
app.use(express.logger({stream: accessLogfile}));
```

至于错误日志，需要单独实现错误响应，修改如下：

```
app.configure('production', function(){
app.error(function (err, req, res, next) {
var meta = '[' + new Date() + '] ' + req.url + '\n';
   errorLogfile.write(meta + err.stack + '\n');
   next();
  });
});
```

这段代码的功能是通过app.error注册错误响应函数，在其中将错误写入错误日志流。

现在重新运行服务器，在浏览器中访问 http://127.0.0.1:3000/，即可在app.js 同一目录下的 access.log 文件中看到与以下类似的内容：

```
    127.0.0.1 - - [Thu, 5 Apr 2012 15:29:28 GMT] "GET / HTTP/1.1" 200 3389 "-" "Mozilla/5.0
(Macintosh; Intel Mac OS X 10_7_3) AppleWebKit/535.19 (KHTML, like Gecko)
Chrome/18.0.1025.162 Safari/535.19"
    127.0.0.1 - - [Thu, 5 Apr 2012 15:29:28 GMT] "GET /stylesheets/bootstrap-responsive.css
HTTP/1.1" 304 - "http://127.0.0.1:3000/" "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_7_3)
AppleWebKit/535.19 (KHTML, like Gecko) Chrome/18.0.1025.162 Safari/535.19"
    127.0.0.1 - - [Thu, 5 Apr 2012 15:29:28 GMT] "GET /javascripts/jquery.js HTTP/1.1" 304
- "http://127.0.0.1:3000/" "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_7_3)
AppleWebKit/535.19 (KHTML, like Gecko) Chrome/18.0.1025.162 Safari/535.19"
    127.0.0.1 - - [Thu, 5 Apr 2012 15:29:28 GMT] "GET /javascripts/bootstrap.js HTTP/1.1"
304 - "http://127.0.0.1:3000/" "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_7_3)
AppleWebKit/535.19 (KHTML, like Gecko) Chrome/18.0.1025.162 Safari/535.19"
```

```
    127.0.0.1 - - [Thu, 5 Apr 2012 15:29:28 GMT] "GET /stylesheets/bootstrap.css HTTP/1.1"
304 - "http://127.0.0.1:3000/" "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_7_3)
AppleWebKit/535.19 (KHTML, like Gecko) Chrome/18.0.1025.162 Safari/535.19"
    127.0.0.1 - - [Thu, 5 Apr 2012 15:29:28 GMT] "GET /favicon.ico HTTP/1.1" 404 - "-"
"Mozilla/5.0 (Macintosh; Intel Mac OS X 10_7_3) AppleWebKit/535.19 (KHTML, like Gecko)
Chrome/18.0.1025.162 Safari/535.19"
```

为了产生一个错误，我们修改routes/index.js 中“/”的响应函数，加入以下代码：

```
throw new Error('An error for test purposes.');
```

再次访问 http://127.0.0.1:3000/，可以看到 error.log 输出了完整的错误内容，如下所示：

```
[Thu Apr 5 2012 23:33:21 GMT+0800 (CST)] /
Error: An error for test purposes.
    at Object.<anonymous> (/byvoid/microblog/routes/index.js:7:11)
    at nextMiddleware (/byvoid/microblog/node_modules/express/node_modules/connect/
      lib/middleware/router.js:175:25)
    at param (/byvoid/microblog/node_modules/express/node_modules/connect/lib/
      middleware/router.js:183:16)
    at pass (/byvoid/microblog/node_modules/express/node_modules/connect/lib/
      middleware/router.js:191:10)
    at Object.router [as handle] (/byvoid/microblog/node_modules/express/
      node_modules/connect/lib/middleware/router.js:197:6)
    at next (/byvoid/microblog/node_modules/express/node_modules/connect/lib/
      http.js:203:15)
    at /byvoid/microblog/node_modules/express/node_modules/connect/lib/middleware/
      session.js:323:9
    at /byvoid/microblog/node_modules/express/node_modules/connect/lib/middleware/
      session.js:342:9
    at /byvoid/microblog/node_modules/connect-mongo/lib/connect-mongo.js:151:15
    at /byvoid/microblog/node_modules/connect-mongo/node_modules/mongodb/lib/
      mongodb/collection.js:885:34
```

这样，一个简单有效的日志功能就这样实现了。

6.3.2　使用`cluster`模块

从0.6 版本开始，Node.js 提供了一个核心模块：`cluster`。`cluster`的功能是生成与当前进程相同的子进程，并且允许父进程和子进程之间共享端口。Node.js 的另一个核心模块`child_process`也提供了相似的进程生成功能，但最大的区别在于`cluster`允许跨进程端口复用，给我们的网络服务器开发带来了很大的方便。

为了在外部模块调用app.js，首先需要禁止服务器自动启动。修改app.js，在`app.listen(3000);`前后加上判断语句：

```
if (!module.parent) {
  app.listen(3000);
  console.log("Express server listening on port %d in %s mode", app.address().port,
    app.settings.env);
}
```

这个语句的功能是判断当前模块是不是由其他模块调用的，如果不是，说明它是直接启动的，此时启动调试服务器；如果是，则不自动启动服务器。经过这样的修改，以后直接调用node app.js服务器会直接运行，但在其他模块中调用require('./app')则不会自动启动，需要再显式地调用 listen() 函数。

接下来就让我们通过cluster 调用app.js。创建cluster.js，内容如下所示：

```
var cluster = require('cluster');
var os = require('os');

// 获取CPU 的数量
var numCPUs = os.cpus().length;

var workers = {};
if (cluster.isMaster) {
  // 主进程分支
cluster.on('death', function (worker) {
    // 当一个工作进程结束时，重启工作进程
delete workers[worker.pid];
    worker = cluster.fork();
    workers[worker.pid] = worker;
  });
  // 初始开启与CPU 数量相同的工作进程
for (var i = 0; i < numCPUs; i++) {
var worker = cluster.fork();
    workers[worker.pid] = worker;
  }
} else {
  // 工作进程分支，启动服务器
var app = require('./app');
app.listen(3000);
}
// 当主进程被终止时，关闭所有工作进程
process.on('SIGTERM', function () {
for (var pid in workers) {
   process.kill(pid);
  }
process.exit(0);
});
```

cluster.js的功能是创建与CPU核心个数相同的服务器进程，以确保充分利用多核CPU的资源。主进程生成若干个工作进程，并监听工作进程结束事件，当工作进程结束时，重新启动一个工作进程。分支进程产生时会自顶向下重新执行当前程序，并通过分支判断进入工作进程分支，在其中读取模块并启动服务器。通过cluster启动的工作进程可以直接实现端口复用，因此所有工作进程只需监听同一端口。当主进程终止时，还要主动关闭所有工作进程。

在终端中执行node cluster.js 命令，可以看到进程列表中启动了多个 node进程（8核CPU）：

```
12408 ?        00:01:28 node
12411 ?        00:01:27 node
12412 ?        00:01:28 node
12414 ?        00:01:31 node
12416 ?        00:01:34 node
12418 ?        00:01:44 node
12420 ?        00:01:38 node
12422 ?        00:01:34 node
12424 ?        00:02:14 node
```

终止工作进程，新的工作进程会立即启动，终止主进程，所有工作进程也会同时结束。这样，一个既能利用多核资源，又有实现故障恢复功能的服务器就诞生了。

6.3.3 启动脚本

接下来，我们还需要一个启动脚本来简化维护工作。如果你维护过Linux 服务器，会对 /etc/init.d/ 下面的脚本有印象。例如使用/etc/init.d/nginx start 和/etc/init.d/nginx stop 可以启动和关闭Nginx 服务器。我们通过bash 脚本也来实现一个类似的功能，创建microblog 并使用chmod +x microblog 赋予其执行权限，脚本内容为：

```
#! /bin/sh

NODE_ENV=production
DAEMON="node cluster.js"
NAME=Microblog
DESC=Microblog
PIDFILE="microblog.pid"

case "$1" in
  start)
        echo "Starting $DESC: "
              nohup $DAEMON > /dev/null &
        echo $! > $PIDFILE
        echo "$NAME."
              ;;
stop)
        echo "Stopping $DESC: "
              pid='cat $PIDFILE'
        kill $pid
              rm $PIDFILE
        echo "$NAME."
              ;;
esac

exit 0
```

它的功能是通过nohup 启动服务器，使进程不会因为退出终端而关闭，同时将主进程的pid 写入microblog.pid 文件。当调用结束命令时，从microblog.pid 读取pid 的值，终止主进程以关闭服务器。

运行./microblog start，结果如下：

```
Starting Microblog:
Microblog.
```

在该目录下生成了microblog.pid文件，查看进程表可以发现服务器已经启动。关闭服务器时只需执行./microblog stop，即可结束所有工作进程。

有了这个启动脚本，我们就可以实现服务器的开机自动启动了，根据不同的操作系统，将其加入启动运行项即可，唯一需要修改的地方是DAEMON 和PIDFILE 应该写成绝对路径，以便在不同的目录下运行。

警告 这段脚本只支持 POSIX 操作系统，如 Linux、Mac OS 等，在 Windows 下不可用。

6.3.4 共享 80 端口

到目前为止，网站都是运行在3000端口下的，也就是说用户必须在网址中加入:3000才能访问网站。默认的HTTP 端口是80，因此必须监听80端口才能使网址更加简洁。如果整个服务器只有一个网站，那么只需让app.js 监听80 端口即可。但很多时候一个服务器上运行着不止一个网站，尤其是还有用其他语言（如PHP）写成的网站，这该怎么办呢？此时虚拟主机可以粉墨登场了。

虚拟主机，就是让多个网站共享使用同一服务器同一IP地址，通过域名的不同来划分请求。主流的HTTP服务器都提供了虚拟主机支持，如Nginx、Apache、IIS等。我们以Nginx为例，介绍如何通过反向代理实现Node.js 虚拟主机。

在Nginx 中设置反向代理和虚拟主机非常简单，下面是配置文件的一个示例：

```
server {
    listen 80;
    server_name mysite.com;

    location / {
    proxy_pass http://localhost:3000;
    }
}
```

这个配置文件的功能是监听访问mysite.com 80 端口的请求，并将所有的请求转发给http://localhost:3000，即我们的Node.js 服务器。现在访问http://mysite.com/，就相当于服务器

访问http://localhost:3000了。

在添加了虚拟主机以后，还可以在Nginx配置文件中添加访问静态文件的规则（具体请参考Nginx文档），删去app.js中的`app.use(express.static(__dirname + '/public'));`。这样可以直接让Nginx 来处理静态文件，减少反向代理以及Node.js 的开销。

6.4　Node.js 不是银弹

在本书正文的最后一节，我们打算讨论一下Node.js 不适合做什么，涉及它的不足之处和一些弊端。

在西方古老的传说里，有一种叫做“狼人”的可怕生物。这种生物平时和人类没有什么不同之处，但每到月圆之夜，他们就会变成狼身。当他们变成狼以后，兽性会不能控制，开始袭击普通的人类。狼人给人类带来了巨大的恐惧，因为他们是无法被一般的手段杀死的，只有用赐福过的银弹（Silver Bullet）才能杀死狼人。“银弹”因此成为了“任何能够带来极大效果的直接解决方案”的代名词。

Fred Brooks①在1987 年发表了一篇关于软件工程的经典文章——“No Silver Bullet”（没有银弹）。所谓的“没有银弹”指的就是没有任何一项技术或方法可使软件工程的生产力能像摩尔定律一样在十年内提高超过十倍，不仅当时没有，现在也没有，今后也不会有。这篇文章收录在《人月神话》（*The Mythical Man-Month*）一书中，被誉为软件工程领域的基本定律之一。

Node.js 也不例外，它不是什么能够大幅度提高软件开发效率和质量的灵丹妙药。无论使用什么语言、工具，所能改变的仅仅是开发的舒适程度和方便程度，而最终软件的好坏所能改变的范围相当有限。任何试图以限制程序员犯错来提高软件质量的方式最终都已经以失败告终。真正优秀的软件是靠优秀的程序员开发出来的，优秀的语言、平台、工具只有在优秀的程序员的手中才能显现出它的威力。

Node.js 不适合做什么

Node.js 是一个优秀的平台，吸引大量开发者关注。它有许多传统架构不具备的优点，以至于我们情不自禁地愿意用它来做开发。Node.js 和任何东西一样，都有它擅长的和不擅长的事情，如果你非要用它来做它不擅长的事情，那么你将会陷入僵局之中。尽管你可以以喜欢、它很新潮、性能高为借口，却不得不写出难看的代码。

和大多数新技术的本质一样，Node.js 也只是旧瓶盛新酒。大多数人事实上并不知道为什么使用Node.js，只是因为你了解它，所以使用它，进而觉得它好，觉得它是最合适的。这是一个必须跳出的误区，否则你就像是得了强迫症，不管三七二十一，遇到什么问题都用

① IBM 大型计算机之父，曾经开发过OS/360 等大型计算机的操作系统。

Node.js 解决。

所以现在就让我们来谈谈 Node.js 不适合做的事情吧。

1. 计算密集型的程序

在Node.js 0.8 版本之前，Node.js 不支持多线程。当然，这是一种设计哲学问题，因为Node.js的开发者和支持者坚信单线程和事件驱动的异步式编程比传统的多线程编程运行效率更高。但事实上多线程可以达到同样的吞吐量，尽管可能开销不小，但不必为多核环境进行特殊的配置。相比之下，Node.js 由于其单线程性的特性，必须通过多进程的方法才能充分利用多核资源。

理想情况下，Node.js单线程在执行的过程中会将一个CPU核心完全占满，所有的请求必须等待当前请求处理完毕以后进入事件循环才能响应。如果一个应用是计算密集型的，那么除非你手动将它拆散，否则请求响应延迟将会相当大。例如，某个事件的回调函数中要进行复杂的计算，占用CPU 200毫秒，那么事件循环中所有的请求都要等待200毫秒。为了提高响应速度，你唯一的办法就是把这个计算密集的部分拆成若干个逻辑，这给编程带来了额外的复杂性。即使这样，系统的总吞吐量和总响应延迟也不会降低，只是调度稍微公平了一些。

不过好在真正的Web 服务器中，很少会有计算密集的部分，如果真的有，那么它不应该被实现为即时的响应。正确的方式是给用户一个提示，说服务器正在处理中，完成后会通知用户，然后交给服务器的其他进程甚至其他专职的服务器来做这件事。

2. 单用户多任务型应用

前面我们讨论的通常都是服务器端编程，其中一个假设就是用户数量很多。但如果面对的是单用户，譬如本地的命令行工具或者图形界面，那么所谓的大量并发请求就不存在了。于是另一个恐怖的问题出现了，尽管是单用户，却不一定是单任务。例如给用户提供界面的同时后台在进行某个计算，为了让用户界面不出现阻塞状态，你不得不开启多线程或多进程。而Node.js 线程或进程之间的通信到目前为止还很不便，因为它根本没有锁，因而号称不会死锁。Node.js 的多进程往往是在执行同一任务，通过多进程利用多处理器的资源，但遇到多进程相互协作时，就显得捉襟见肘了。

3. 逻辑十分复杂的事务

Node.js 的控制流不是线性的，它被一个个事件拆散，但人的思维却是线性的，当你试图转换思维来迎合语言或编译器时，就不得不作出牺牲。举例来说，你要实现一个这样的逻辑：从银行取钱，拿钱去购买某个虚拟商品，买完以后加入库存数据库，这中间的任何一步都可能会涉及数十次的I/O操作，而且任何一次操作失败以后都要进行回滚操作。这个过程是线性的，已经很复杂了，如果要拆分为非线性的逻辑，那么其复杂程度很可能就达到无法维护的地步了。

Node.js更善于处理那些逻辑简单但访问频繁的任务，而不适合完成逻辑十分复杂的工作。

6

4. Unicode 与国际化

Node.js 不支持完整的Unicode，很多字符无法用`string` 表示。公平地说这不是Node.js 的缺陷，而是JavaScript 标准的问题。目前JavaScript 支持的字符集还是双字节的UCS2，即用两个字节来表示一个Unicode 字符，这样能表示的字符数量是65536。显然，仅仅是汉字就不止这个数目，很多生僻汉字，以及一些较为罕见语言的文字都无法表示。这其实是一个历史遗留问题，像2000 年问题（俗称千年虫）一样，都起源于当时人们的主观判断。最早的Unicode 设计者认为65536个字符足以囊括全世界所有的文字了，因此那个时候盲目兼容Unicode 的系统或平台（如Windows、Java 和JavaScript）在后来都遇到了问题。

Unicode 随后意识到2个字节是不够的，因此推出了UCS4，即用4 个字节来表示一个Unicode 字符。很多原先用定长编码的UCS2 的系统都升级为了变长编码的UTF-16，因为只有它向下兼容UCS2。UTF-16 对UCS2 以内的字符采用定长的双字节编码，而对它以外的部分使用多字节的变长编码。这种方式的好处是在绝大多数情况下它都是定长的编码，有利于提高运算效率，而且兼容了UCS2，但缺点是它本质还是变长编码，程序中处理多少有些不便。

许多号称支持UTF-16 的平台仍然只支持它的子集UCS2，而不支持它的变长编码部分。相比之下，UTF-8 完全是变长编码，有利于传输，而UTF-32 或UCS4 则是4 字节的定长编码，有利于计算。

当下的JavaScript 内部支持的仍是定长的UCS2 而不是变长的UTF-16，因此对于处理UCS4 的字符它无能为力。所有的JavaScript 引擎都被迫保留了这个缺陷，包括V8 在内，因此你无法使用Node.js 处理罕见的字符。想用Node.js 实现一个多语言的字典工具？还是算了吧，除非你放弃使用`string` 数据类型，把所有的字符当作二进制的`Buffer` 数据来处理。

6.5　参考资料

- “深入浅出Node.js （三）：深入Node.js 的模块机制”: http://www.infoq.com/cn/articles/nodejs-module-mechanism。
- 《Node Web开发》David Herron著，人民邮电出版社出版。
- “遭遇回调函数产生的陷阱”: http://club.cnodejs.org/topic/4f6f057f8a04d82a3d0d230a。
- *What Is Node? JavaScript Breaks Out of the Browser*: http://shop.oreilly.com/product/0636920021506. do。
- “Node.js 究竟是什么？一个“编码就绪”服务器”: http://www.ibm.com/developerworks/cn/opensource/os-nodejs/。
- “Node.js is Cancer”: http://teddziuba.com/2011/10/node-js-is-cancer.html。
- “Node.js is Candy”: http://xentac.net/2011/10/05/1-nodejs-is-candy.html。
- “V8 does not support UCS 4 characters”: http://code.google.com/p/v8/issues/ detail?id=1697。

JavaScript的高级特性

附录 A

长久以来，JavaScript 总是被广大的专业开发者轻视，不少人以为 JavaScript 是像 VBScript 一样的雕虫小技，或者说是给非专业的网页设计者用的简易工具。而早期的因特网上也恰恰流传着大量低质量的 JavaScript 代码，很多都是可视化网页设计工具生成的，复杂而混乱，更加深了人们对它的不良印象。在当时，JavaScript 的一个主要作用是在网页上显示出花哨的效果，譬如弹出令人厌烦的广告窗口。

早期的 JavaScript 运行效率低下、浏览器兼容性问题严重。就连 JavaScript 之父 Brendan Eich 都觉得它很烂，从来没有想过它能够发展成今天的样子。后来随着以 Gmail 为代表的 Web 2.0 应用的兴起，人们开始重新认识 JavaScript。

JavaScript 经历了一个十分纠结的发展过程，因为 ECMAScript 新标准总是在提出后若干年才会被浏览器开发商陆续实现，所以开发者不得不忍痛割爱放弃许多 JavaScript 优美的新特性，以保持浏览器之间的兼容性。值得庆幸的是，这些问题在 Node.js 中已不复存在，我们可以放心地享受 JavaScript 的全部特性给我们带来的便利了。这些特性大多已经是现代编程语言共有的理念，例如面向对象、函数式编程思想、lambda 演算、闭包、动态绑定等。

我假设你了解 JavaScript 的基本语法，并且对面向对象的语言有一定的理解，如果你还知道函数式编程（functional programming），那么你将可以很容易地理解闭包。本附录通过大量的示例帮你了解 JavaScript 众多特性，理解 JavaScript 背后的机制。我们以作用域、闭包和对象为线索，介绍 JavaScript 编程中常用到的特性和技巧。

A.1　作用域

作用域（scope）是结构化编程语言中的重要概念，它决定了变量的可见范围和生命周期，正确使用作用域可以使代码更清晰、易懂。作用域可以减少命名冲突，而且是垃圾回收的基本单元。和 C、C++、Java 等常见语言不同，JavaScript 的作用域不是以花括号包围的块级作用域（block scope），这个特性经常被大多数人忽视，因而导致莫名其妙的错误。例如下面代码，在大多数类 C 的语言中会出现变量未定义的错误，而在 JavaScript 中却完全合法：

```
if (true) {
  var somevar = 'value';
}
console.log(somevar); // 输出 value
```

这是因为 JavaScript 的作用域完全是由函数来决定的，`if`、`for` 语句中的花括号不是独立的作用域。

A.1.1　函数作用域

不同于大多数类 C 的语言，由一对花括号封闭的代码块就是一个作用域，JavaScript 的

作用域是通过函数来定义的，在一个函数中定义的变量只对这个函数内部可见，我们称为函数作用域。在函数中引用一个变量时，JavaScript 会先搜索当前函数作用域，或者称为“局部作用域”，如果没有找到则搜索其上层作用域，一直到全局作用域。我们看一个简单的例子：

```
var v1 = 'v1';

var f1 = function() {
  console.log(v1); // 输出 v1
};
f1();

var f2 = function() {
  var v1 = 'local';
  console.log(v1); // 输出 local
};
f2();
```

以上示例十分明了，JavaScript 的函数定义是可以嵌套的，每一层是一个作用域，变量搜索顺序是从内到外。下面这个例子可能就有些令人困惑：

```
var scope = 'global';

var f = function() {
  console.log(scope); // 输出 undefined
  var scope = 'f';
}
f();
```

上面代码可能和你预想的不一样，没有输出 global，而是undefined，这是为什么呢？这是 JavaScript 的一个特性，按照作用域搜索顺序，在 console.log 函数访问 scope 变量时，JavaScript 会先搜索函数 f 的作用域，恰巧在 f 作用域里面搜索到 scope 变量，所以上层作用域中定义的 scope 就被屏蔽了，但执行到 console.log 语句时，scope 还没被定义，或者说初始化，所以得到的就是 undefined 值了。

我们还可以从另一个角度来理解：对于开发者来说，在访问未定义的变量或定义了但没有初始化的变量时，获得的值都是 undefined。于是我们可以认为，无论在函数内什么地方定义的变量，在一进入函数时就被定义了，但直到 var 所在的那一行它才被初始化，所以在这之前引用到的都是 undefined 值。（事实上，JavaScript 的内部实现并不是这样，未定义变量和值为 undefined 的变量还是有区别的。）

函数作用域的嵌套

接下来看一个稍微复杂的例子：

```
var f = function() {
  var scope = 'f0';
  (function() {
    var scope = 'f1';
    (function() {
      console.log(scope); // 输出 f1
    })();
  })();
};
f();
```

上面是一个函数作用域嵌套的例子，我们在最内层函数引用了 scope 变量，通过作用域搜索，找到了其父作用域中定义的 scope 变量。

有一点需要注意：函数作用域的嵌套关系是定义时决定的，而不是调用时决定的，也就是说，JavaScript 的作用域是静态作用域，又叫词法作用域，这是因为作用域的嵌套关系可以在语法分析时确定，而不必等到运行时确定。下面的例子说明了这一切：

```
var scope = 'top';

var f1 = function() {
  console.log(scope);
};
f1(); // 输出 top

var f2 = function() {
  var scope = 'f2';
  f1();
};
f2(); // 输出 top
```

这个例子中，通过 f2 调用的 f1 在查找 scope 定义时，找到的是父作用域中定义的 scope 变量，而不是 f2 中定义的 scope 变量。这说明了作用域的嵌套关系不是在调用时确定的，而是在定义时确定的。

A.1.2　全局作用域

在 JavaScript 中有一种特殊的对象称为 *全局对象*。这个对象在Node.js 对应的是 global 对象，在浏览器中对应的是 window 对象。由于全局对象的所有属性在任何地方都是可见的，所以这个对象又称为 *全局作用域*。全局作用域中的变量不论在什么函数中都可以被直接引用，而不必通过全局对象。

满足以下条件的变量属于全局作用域：

- 在最外层定义的变量；
- 全局对象的属性；

- 任何地方隐式定义的变量（未定义直接赋值的变量）。

需要格外注意的是第三点，在任何地方隐式定义的变量都会定义在全局作用域中，即不通过 `var` 声明直接赋值的变量。这一点经常被人遗忘，而模块化编程的一个重要原则就是避免使用全局变量，所以我们在任何地方都不应该隐式定义变量。

A.2　闭包

闭包（closure）是函数式编程中的概念，出现于 20 世纪 60 年代，最早实现闭包的语言是 Scheme，它是 LISP 的一种方言。之后闭包特性被其他语言广泛吸纳。

闭包的严格定义是"由函数（环境）及其封闭的自由变量组成的集合体。"这个定义对于大家来说有些晦涩难懂，所以让我们先通过例子和不那么严格的解释来说明什么是闭包，然后再举例说明一些闭包的经典用途。

A.2.1　什么是闭包

通俗地讲，JavaScript 中每个的函数都是一个闭包，但通常意义上嵌套的函数更能够体现出闭包的特性，请看下面这个例子：

```
var generateClosure = function() {
  var count = 0;
  var get = function() {
    count ++;
    return count;
  };
  return get;
};

var counter = generateClosure();
console.log(counter()); // 输出 1
console.log(counter()); // 输出 2
console.log(counter()); // 输出 3
```

这段代码中，`generateClosure()` 函数中有一个局部变量`count`，初值为 0。还有一个叫做 `get` 的函数，`get` 将其父作用域，也就是 `generateClosure()` 函数中的 `count` 变量增加 1，并返回 `count` 的值。`generateClosure()` 的返回值是 `get` 函数。在外部我们通过 `counter` 变量调用了 `generateClosure()` 函数并获取了它的返回值，也就是 `get` 函数，接下来反复调用几次 `counter()`，我们发现每次返回的值都递增了 1。

让我们看看上面的例子有什么特点，按照通常命令式编程思维的理解，`count` 是 `generateClosure` 函数内部的变量，它的生命周期就是 `generateClosure` 被调用的时期，当 `generateClosure` 从调用栈中返回时，`count` 变量申请的空间也就被释放。问题

是，在 generateClosure() 调用结束后，counter() 却引用了“已经释放了的” count 变量，而且非但没有出错，反而每次调用 counter() 时还修改并返回了 count。这是怎么回事呢？

这正是所谓闭包的特性。当一个函数返回它内部定义的一个函数时，就产生了一个闭包，闭包不但包括被返回的函数，还包括这个函数的定义环境。上面例子中，当函数 generateClosure() 的内部函数 get 被一个外部变量 counter 引用时，counter 和 generateClosure() 的局部变量就是一个闭包。如果还不够清晰，下面这个例子可以帮助你理解：

```
var generateClosure = function() {
  var count = 0;
  var get = function() {
    count ++;
    return count;
  };
  return get;
};

var counter1 = generateClosure();
var counter2 = generateClosure();
console.log(counter1()); // 输出 1
console.log(counter2()); // 输出 1
console.log(counter1()); // 输出 2
console.log(counter1()); // 输出 3
console.log(counter2()); // 输出 2
```

上面这个例子解释了闭包是如何产生的：counter1 和 counter2 分别调用了 generate-Closure() 函数，生成了两个闭包的实例，它们内部引用的 count 变量分别属于各自的运行环境。我们可以理解为，在 generateClosure() 返回 get 函数时，私下将 get 可能引用到的 generateClosure() 函数的内部变量（也就是 count 变量）也返回了，并在内存中生成了一个副本，之后 generateClosure() 返回的函数的两个实例 counter1 和 counter2 就是相互独立的了。

A.2.2　闭包的用途

1. 嵌套的回调函数

闭包有两个主要用途，一是实现嵌套的回调函数，二是隐藏对象的细节。让我们先看下面这段代码示例，了解嵌套的回调函数。如下代码是在 Node.js 中使用 MongoDB 实现一个简单的增加用户的功能：

```
exports.add_user = function(user_info, callback) {
  var uid = parseInt(user_info['uid']);
  mongodb.open(function(err, db) {
    if (err) {callback(err); return;}
    db.collection('users', function(err, collection) {
      if (err) {callback(err); return;}
      collection.ensureIndex("uid", function(err) {
        if (err) {callback(err); return;}
        collection.ensureIndex("username", function(err) {
          if (err) {callback(err); return;}
          collection.findOne({uid: uid}, function(err) {
            if (err) {callback(err); return;}
            if (doc) {
              callback('occupied');
            } else {
              var user = {
                uid: uid,
                user: user_info,
              };
              collection.insert(user, function(err) {
                callback(err);
              });
            }
          });
        });
      });
    });
  });
};
```

如果你对 Node.js 或 MongoDB 不熟悉，没关系，不需要去理解细节，只要看清楚大概的逻辑即可。这段代码中用到了闭包的层层嵌套，每一层的嵌套都是一个回调函数。回调函数不会立即执行，而是等待相应请求处理完后由请求的函数回调。我们可以看到，在嵌套的每一层中都有对 `callback` 的引用，而且最里层还用到了外层定义的 `uid` 变量。由于闭包机制的存在，即使外层函数已经执行完毕，其作用域内申请的变量也不会释放，因为里层的函数还有可能引用到这些变量，这样就完美地实现了嵌套的异步回调。

提示 尽管可以这么做，上面这种回调函数深层嵌套的实现并不优美，本书第 6 章中介绍了控制流优化的方法。

2. 实现私有成员

我们知道，JavaScript 的对象没有私有属性，也就是说对象的每一个属性都是曝露给外部的。这样可能会有安全隐患，譬如对象的使用者直接修改了某个属性，导致对象内部数据的一致性受到破坏等。JavaScript通过约定在所有私有属性前加上下划线（例如`_myPrivateProp`），

表示这个属性是私有的，外部对象不应该直接读写它。但这只是个非正式的约定，假设对象的使用者不这么做，有没有更严格的机制呢？答案是有的，通过闭包可以实现。让我们再看看前面那个例子：

```
var generateClosure = function() {
  var count = 0;
  var get = function() {
    count ++;
    return count;
  };
  return get;
};

var counter = generateClosure();
console.log(counter()); // 输出 1
console.log(counter()); // 输出 2
console.log(counter()); // 输出 3
```

我们可以看到，只有调用 `counter()` 才能访问到闭包内的 `count` 变量，并按照规则对其增加1，除此之外决无可能用其他方式找到 `count` 变量。受到这个简单例子的启发，我们可以把一个对象用闭包封装起来，只返回一个“访问器”的对象，即可实现对细节隐藏。关于实现JavaScript对象私有成员的更多信息，请参考http://javascript.crockford.com/private.html。

A.3　对象

提起面向对象的程序设计语言，立刻让人想起的是 C++、Java 等这类静态强类型语言，以及 Python、Ruby 等脚本语言，它们共有的特点是基于类的面向对象。而说到 JavaScript，很少能让人想到它面向对象的特性，甚至有人说它不是面向对象的语言，因为它没有类。没错，JavaScript 真的没有类，但 JavaScript 是面向对象的语言。JavaScript 只有对象，对象就是对象，不是类的实例。

因为绝大多数面向对象语言中的对象都是基于类的，所以经常有人混淆类的实例与对象的概念。对象就是类的实例，这在大多数语言中都没错，但在 JavaScript 中却不适用。JavaScript 中的对象是基于原型的，因此很多人在初学 JavaScript 对象时感到无比困惑。通过这一节，我们将重新认识 JavaScript 中对象，充分理解基于原型的面向对象的实质。

A.3.1　创建和访问

JavaScript 中的对象实际上就是一个由属性组成的关联数组，属性由名称和值组成，值的类型可以是任何数据类型，或者函数和其他对象。注意 JavaScript 具有函数式编程的特性，所以函数也是一种变量，大多数时候不用与一般的数据类型区分。

在 JavaScript 中，你可以用以下方法创建一个简单的对象：

```
var foo = {};
foo.prop_1 = 'bar';
foo.prop_2 = false;
foo.prop_3 = function() {
    return 'hello world';
}
console.log(foo.prop_3());
```

以上代码中，我们通过 `var foo = {};` 创建了一个对象，并将其引用赋值给 `foo`，通过 `foo.prop1` 来获取它的成员并赋值，其中 `{}` 是对象字面量的表示方法，也可以用 `var foo = new Object()` 来显式地创建一个对象。

1. 使用关联数组访问对象成员

我们还可以用关联数组的模式来创建对象，以上代码修改为：

```
var foo = {};
foo['prop1'] = 'bar';
foo['prop2'] = false;
foo['prop3'] = function() {
  return 'hello world';
}
```

在 JavaScript 中，使用句点运算符和关联数组引用是等价的，也就是说任何对象（包括 `this` 指针）都可以使用这两种模式。使用关联数组的好处是，在我们不知道对象的属性名称的时候，可以用变量来作为关联数组的索引。例如：

```
var some_prop = 'prop2';
foo[some_prop] = false;
```

2. 使用对象初始化器创建对象

上述的方法只是让你对JavaScript对象的定义有个了解，真正在使用的时候，我们会采用下面这种更加紧凑明了的方法：

```
var foo = {
  'prop1': 'bar',
  prop2: 'false',
  prop3: function (){
    return 'hello world';
  }
};
```

这种定义的方法称为对象的初始化器。注意，使用初始化器时，对象属性名称是否加引号是可选的，除非属性名称中有空格或者其他可能造成歧义的字符，否则没有必要使用引号。

A.3.2 构造函数

前一小节讲述的对象创建方法都有一个弱点，就是创建对象的代码是一次性的。如果我们想创建多个规划好的对象，有若干个固定的属性、方法，并能够初始化，就像 C++ 语言中的对象一样，该如何做呢？别担心，JavaScript 提供了构造函数，让我们来看看应该如何创建复杂的对象。

```
function User(name, uri) {
  this.name = name;
  this.uri = uri;
  this.display = function() {
    console.log(this.name);
  }
}
```

以上是一个简单的构造函数，接下来用 `new` 语句来创建对象：

```
var someuser = new User('byvoid', 'http://www.byvoid.com');
```

然后就可以通过 `someuser` 来访问这个对象的属性和方法了。

A.3.3 上下文对象

在 JavaScript 中，上下文对象就是 `this` 指针，即被调用函数所处的环境。上下文对象的作用是在一个函数内部引用调用它的对象本身，JavaScript 的任何函数都是被某个对象调用的，包括全局对象，所以 `this` 指针是一个非常重要的东西。

提示 JavaScript 中并没有像 C++ 一样的指针概念，这里所谓的 `this` 指针只是沿用习惯的说法而已。

在前面使用构造函数的代码中我们已经看到了 `this` 的使用方法，下面代码可以更佳清楚地说明上下文对象的使用方式：

```
var someuser = {
  name: 'byvoid',
  display: function() {
    console.log(this.name);
  }
};

someuser.display(); // 输出 byvoid

var foo = {
```

```
  bar: someuser.display,
  name: 'foobar'
};

foo.bar(); // 输出 foobar
```

JavaScript 的函数式编程特性使得函数可以像一般的变量一样赋值、传递和计算，我们看到在上面代码中，foo 对象的 bar 属性是 someuser.display 函数，使用 foo.bar() 调用时，bar 和 foo 对象的函数看起来没有区别，其中的 this 指针不属于某个函数，而是函数调用时所属的对象。

在 JavaScript 中，本质上，函数类型的变量是指向这个函数实体的一个引用，在引用之间赋值不会对对象产生复制行为。我们可以通过函数的任何一个引用调用这个函数，不同之处仅仅在于上下文。下面例子可以帮助我们理解：

```
var someuser = {
  name: 'byvoid',
  func: function() {
    console.log(this.name);
  }
};

var foo = {
  name: 'foobar'
};

someuser.func(); // 输出 byvoid

foo.func = someuser.func;
foo.func(); // 输出 foobar

name = 'global';
func = someuser.func;
func(); // 输出 global
```

仔细观察上面的例子，使用不同的引用来调用同一个函数时，this 指针永远是这个引用所属的对象。在前面的章节中我们提到了 JavaScript 的函数作用域是静态的，也就是说一个函数的可见范围是在预编译的语法分析中就可以确定的，而上下文对象则可以看作是静态作用域的补充。

1. call 和 apply

在 JavaScript 中，call 和 apply 是两个神奇的方法，但同时也是容易令人迷惑的两个方法，乃至许多对 JavaScript 有经验的人也不太清楚它们的用法。call 和 apply 的功能是以不同的对象作为上下文来调用某个函数。简而言之，就是允许一个对象去调用另一个对象的成员函数。乍一看似乎很不可思议，而且容易引起混乱，但其实 JavaScript 并没有严格的

所谓“成员函数”的概念，函数与对象的所属关系在调用时才展现出来。灵活使用 call 和 apply 可以节省不少时间，在后面我们可以看到，call 可以用于实现对象的继承。

call 和 apply 的功能是一致的，两者细微的差别在于 call 以参数表来接受被调用函数的参数，而 apply 以数组来接受被调用函数的参数。call 和 apply 的语法分别是：

```
func.call(thisArg[, arg1[, arg2[, ...]]])
func.apply(thisArg[, argsArray])
```

其中，func 是函数的引用，thisArg 是 func 被调用时的上下文对象，arg1、arg2 或 argsArray 是传入 func 的参数。我们以下面一段代码为例介绍 call 的工作机制：

```
var someuser = {
  name: 'byvoid',
  display: function(words) {
    console.log(this.name + ' says ' + words);
  }
};

var foo = {
  name: 'foobar'
};

someuser.display.call(foo, 'hello'); // 输出 foobar says hello
```

用 Node.js 运行这段代码，我们可以看到控制台输出了 foobar。someuser.display 是被调用的函数，它通过 call 将上下文改变为 foo 对象，因此在函数体内访问 this.name 时，实际上访问的是 foo.name，因而输出了foobar。

2. **bind**

如何改变被调用函数的上下文呢？前面说过，可以用 call 或 apply 方法，但如果重复使用会不方便，因为每次都要把上下文对象作为参数传递，而且还会使代码变得不直观。针对这种情况，我们可以使用 bind 方法来永久地绑定函数的上下文，使其无论被谁调用，上下文都是固定的。bind 语法如下：

```
func.bind(thisArg[, arg1[, arg2[, ...]]])
```

其中 func 是待绑定函数，thisArg 是改变的上下文对象，arg1、arg2 是绑定的参数表。bind 方法返回值是上下文为 thisArg 的 func。通过下面例子可以帮你理解 bind 的使用方法：

```
var someuser = {
  name: 'byvoid',
  func: function() {
    console.log(this.name);
```

```
  }
};

var foo = {
  name: 'foobar'
};

foo.func = someuser.func;
foo.func(); // 输出 foobar

foo.func1 = someuser.func.bind(someuser);
foo.func1(); // 输出 byvoid

func = someuser.func.bind(foo);
func(); // 输出 foobar

func2 = func;
func2(); // 输出 foobar
```

上面代码直接将 foo.func 赋值为 someuser.func，调用 foo.func() 时，this指针为 foo，所以输出结果是 foobar。foo.func1 使用了 bind 方法，将 someuser 作为this指针绑定到 someuser.func，调用 foo.func1() 时，this指针为 someuser，所以输出结果是 byvoid。全局函数 func 同样使用了 bind 方法，将 foo 作为 this 指针绑定到 someuser.func，调用 func() 时，this 指针为 foo，所以输出结果是 foobar。而 func2 直接将绑定过的 func 赋值过来，与 func 行为完全相同。

3. 使用 bind 绑定参数表

bind 方法还有一个重要的功能：绑定参数表，如下例所示。

```
var person = {
  name: 'byvoid',
  says: function(act, obj) {
    console.log(this.name + ' ' + act + ' ' + obj);
  }
};

person.says('loves', 'diovyb'); // 输出 byvoid loves diovyb

byvoidLoves = person.says.bind(person, 'loves');
byvoidLoves('you'); // 输出 byvoid loves you
```

可以看到，byvoidLoves 将 this 指针绑定到了 person，并将第一个参数绑定到 loves，之后在调用 byvoidLoves 的时候，只需传入第三个参数。这个特性可以用于创建一个函数的“捷径”，之后我们可以通过这个“捷径”调用，以便在代码多处调用时省略重复输入相同的参数。

4. 理解 bind

尽管 bind 很优美，还是有一些令人迷惑的地方，例如下面的代码：

```
var someuser = {
  name: 'byvoid',
  func: function () {
    console.log(this.name);
  }
};

var foo = {
  name: 'foobar'
};

func = someuser.func.bind(foo);
func(); // 输出 foobar

func2 = func.bind(someuser);
func2(); // 输出 foobar
```

全局函数 func 通过someuser.func.bind将this指针绑定到了foo，调用func()输出了foobar。我们试图将func2赋值为已绑定的func重新通过bind将this指针绑定到someuser的结果，而调用func2时却发现输出值仍为foobar，即 this 指针还是停留在 foo 对象上，这是为什么呢？要想解释这个现象，我们必须了解 bind 方法的原理。

让我们看一个 bind 方法的简化版本（不支持绑定参数表）：

```
someuser.func.bind = function(self) {
  return this.call(self);
};
```

假设上面函数是 someuser.func 的 bind 方法的实现，函数体内 this 指向的是 someuser.func，因为函数也是对象，所以 this.call(self) 的作用就是以 self 作为 this指针调用 someuser.func。

```
//将func = someuser.func.bind(foo)展开：
func = function() {
  return someuser.func.call(foo);
};

//再将func2 = func.bind(someuser)展开：
func2 = function() {
  return func.call(someuser);
};
```

从上面展开过程我们可以看出，func2 实际上是以 someuser 作为 func 的this指针调用了 func，而 func 根本没有使用 this 指针，所以两次 bind 是没有效果的。

A.3.4　原型

原型是 JavaScript 面向对象特性中重要的概念，也是大家太熟悉的概念。因为在绝大多数的面向对象语言中，对象是基于类的（例如 Java 和 C++），对象是类实例化的结果。而在 JavaScript 语言中，没有类的概念[①]，对象由对象实例化。打个比方来说，基于类的语言中类就像一个模具，对象由这个模具浇注产生，而基于原型的语言中，原型就好像是一件艺术品的原件，我们通过一台 100% 精确的机器把这个原件复制出很多份。

前面小节的例子中都没有涉及原型，仅仅通过构造函数和 `new` 语句生成类，让我们看看如何使用原型和构造函数共同生成对象。

```
function Person() {
}
Person.prototype.name = 'BYVoid';
Person.prototype.showName = function () {
  console.log(this.name);
};

var person = new Person();
person.showName();
```

上面这段代码使用了原型而不是构造函数初始化对象。这样做与直接在构造函数内定义属性有什么不同呢？

- 构造函数内定义的属性继承方式与原型不同，子对象需要显式调用父对象才能继承构造函数内定义的属性。
- 构造函数内定义的任何属性，包括函数在内都会被重复创建，同一个构造函数产生的两个对象不共享实例。
- 构造函数内定义的函数有运行时闭包的开销，因为构造函数内的局部变量对其中定义的函数来说也是可见的。

下面这段代码可以验证以上问题：

```
function Foo() {
  var innerVar = 'hello';
  this.prop1 = 'BYVoid';
  this.func1 = function(){
    innerVar = '';
  };
}
Foo.prototype.prop2 = 'Carbo';
Foo.prototype.func2 = function () {
  console.log(this.prop2);
```

① 很多时候对象的构造函数会被称为“类”，但实际上并不是严格意义上的类。

```
};

var foo1 = new Foo();
var foo2 = new Foo();

console.log(foo1.func1 == foo2.func1); // 输出 false
console.log(foo1.func2 == foo2.func2); // 输出 true
```

尽管如此，并不是说在构造函数内创建属性不好，而是两者各有适合的范围。那么我们什么时候使用原型，什么时候使用构造函数内定义来创建属性呢？

- 除非必须用构造函数闭包，否则尽量用原型定义成员函数，因为这样可以减少开销。
- 尽量在构造函数内定义一般成员，尤其是对象或数组，因为用原型定义的成员是多个实例共享的。

接下来，我们介绍一下JavaScript中的原型链机制。

原型链

JavaScript 中有两个特殊的对象： `Object` 与 `Function`，它们都是构造函数，用于生成对象。`Object.prototype` 是所有对象的祖先，`Function.prototype` 是所有函数的原型，包括构造函数。我把 JavaScript 中的对象分为三类，一类是用户创建的对象，一类是构造函数对象，一类是原型对象。用户创建的对象，即一般意义上用 `new` 语句显式构造的对象。构造函数对象指的是普通的构造函数，即通过 `new` 调用生成普通对象的函数。原型对象特指构造函数 `prototype` 属性指向的对象。这三类对象中每一类都有一个 `__proto__` 属性，它指向该对象的原型，从任何对象沿着它开始遍历都可以追溯到 `Object.prototype`。构造函数对象有 `prototype` 属性，指向一个原型对象，通过该构造函数创建对象时，被创建对象的 `__proto__` 属性将会指向构造函数的 `prototype` 属性。原型对象有 `constructor` 属性，指向它对应的构造函数。让我们通过下面这个例子来理解原型：

```
function Foo() {
}
Object.prototype.name = 'My Object';
Foo.prototype.name = 'Bar';

var obj = new Object();
var foo = new Foo();
console.log(obj.name); // 输出 My Object
console.log(foo.name); // 输出 Bar
console.log(foo.__proto__.name); // 输出 Bar
console.log(foo.__proto__.__proto__.name); // 输出 My Object
console.log(foo. __proto__.constructor.prototype.name); // 输出 Bar
```

我们定义了一个叫做 `Foo ()`的构造函数，生成了对象 `foo`。同时我们还分别给 `Object` 和 `Foo` 生成原型对象。

图A-1 解析了它们之间错综复杂的关系。

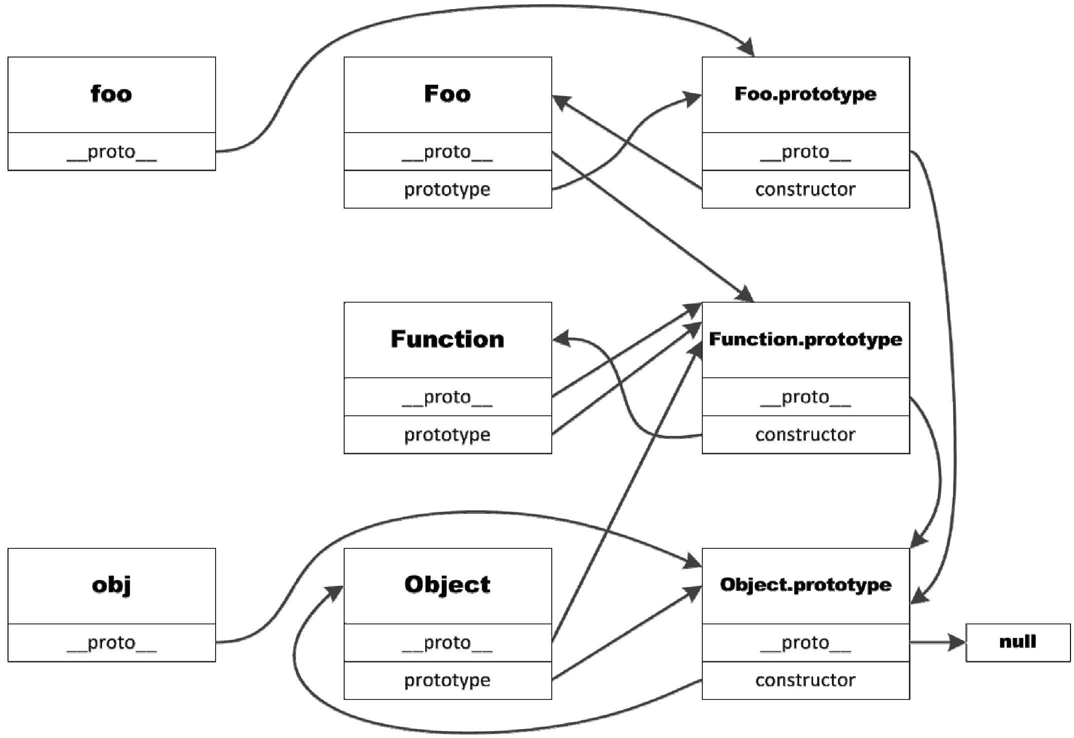


图 A-1　JavaScript 原型之间的关系

在 JavaScript 中，继承是依靠一套叫做原型链（prototype chain）的机制实现的。属性继承的本质就是一个对象可以访问到它的原型链上任何一个原型对象的属性。例如上例的 foo 对象，它拥有 foo. __proto__ 和 foo. __proto__.__proto__ 所有属性的浅拷贝（只复制基本数据类型，不复制对象）。所以可以直接访问foo.constructor（来自foo. __proto__，即Foo.prototype），foo.toString（来自foo. __proto__.__proto__，即Object.prototype）。

A.3.5　对象的复制

JavaScript 和 Java 一样都没有像C语言中一样的指针，所有对象类型的变量都是指向对象的引用，两个变量之间赋值传递一个对象并不会对这个对象进行复制，而只是传递引用。有些时候我们需要完整地复制一个对象，这该如何做呢？ Java 语言中有 clone 方法可以实现对象复制，但 JavaScript 中没有这样的函数。因此我们需要手动实现这样一个函数，一个简单的做法是复制对象的所有属性：

```
Object.prototype.clone = function() {
  var newObj = {};
  for (var i in this) {
    newObj[i] = this[i];
  }
  return newObj;
}

var obj = {
  name: 'byvoid',
  likes: ['node']
};

var newObj = obj.clone();
obj.likes.push('python');

console.log(obj.likes); // 输出 [ 'node', 'python' ]
console.log(newObj.likes); // 输出 [ 'node', 'python' ]
```

上面的代码是一个对象浅拷贝（shallow copy）的实现，即只复制基本类型的属性，而共享对象类型的属性。浅拷贝的问题是两个对象共享对象类型的属性，例如上例中 `likes` 属性指向的是同一个数组。

实现一个完全的复制，或深拷贝（deep copy）并不是一件容易的事，因为除了基本数据类型，还有多种不同的对象，对象内部还有复杂的结构，因此需要用递归的方式来实现：

```
Object.prototype.clone = function() {
  var newObj = {};
  for (var i in this) {
    if (typeof(this[i]) == 'object' || typeof(this[i]) == 'function') {
      newObj[i] = this[i].clone();
    } else {
      newObj[i] = this[i];
    }
  }
  return newObj;
};

Array.prototype.clone = function() {
  var newArray = [];
  for (var i = 0; i < this.length; i++) {
    if (typeof(this[i]) == 'object' || typeof(this[i]) == 'function') {
      newArray[i] = this[i].clone();
    } else {
      newArray[i] = this[i];
    }
  }
  return newArray;
```

```
};

Function.prototype.clone = function() {
  var that = this;
  var newFunc = function() {
    return that.apply(this, arguments);
  };
  for (var i in this) {
    newFunc[i] = this[i];
  }
  return newFunc;
};

var obj = {
  name: 'byvoid',
  likes: ['node'],
  display: function() {
    console.log(this.name);
  },
};

var newObj = obj.clone();
newObj.likes.push('python');
console.log(obj.likes); // 输出 [ 'node' ]
console.log(newObj.likes); // 输出 [ 'node', 'python' ]
console.log(newObj.display == obj.display); // 输出 false
```

上面这个实现看起来很完美，它不仅递归地复制了对象复杂的结构，还实现了函数的深拷贝。这个方法在大多数情况下都很好用，但有一种情况它却无能为力，例如下面的代码：

```
var obj1 = {
  ref: null
};

var obj2 = {
  ref: obj1
};

obj1.ref = obj2;
```

这段代码的逻辑非常简单，就是两个相互引用的对象。当我们试图使用深拷贝来复制 obj1 和 obj2 中的任何一个时，问题就出现了。因为深拷贝的做法是遇到对象就进行递归复制，那么结果只能无限循环下去。对于这种情况，简单的递归已经无法解决，必须设计一套图论算法，分析对象之间的依赖关系，建立一个拓扑结构图，然后分别依次复制每个顶点，并重新构建它们之间的依赖关系。这已经超出了本书的讨论范围，而且在实际的工程操作中几乎不会遇到这种需求，所以我们就不继续讨论了。

Node.js编程规范

附录 B

并没有一个官方的文档规定Node.js应用程序代码的风格，但Node.js代码分割有着一些“事实上的约定”，大多数项目的代码都一定程度上遵循了这一标准。作为Node.js开发新手，我认为有必要遵守这个约定，以便于今后的交流。追根溯源，这个规范发轫于Node.js核心模块的实现，而Node.js核心模块的代码很大程度上符合JavaScript代码的一贯风格。

事实上代码风格永远是一个有争议的话题，没有什么最好的，也不可能被所有人认可。我们在这一章节介绍的规范主要参考了“Felix’s Node.js Style Guide”（http://nodeguide.com/style.html），其中不仅有代码风格上的规范，也有JavaScript特性选择和设计模式上的建议，这些建议都是经验之谈，并非必须遵守，但可能会让你的程序避免很多意外的错误和性能损失。让我们开始介绍吧。

B.1 缩进

在早期的语言规范中，大多数都选择用Tab作为缩进标记，如Delphi、Microsoft C++规范等。但近年来，空格缩进依靠其风格的不变性，逐渐成为了主流。如Python、Ruby甚至C#都采用了空格缩进作为“标准”。但我们选择 两空格 作为Node.js代码的缩进标记，不同于最常见的四空格缩进。这是因为Node.js代码中很容易写出深层的函数嵌套，过多的空格会给阅读带来不便，因此我们选择两空格缩进。

正确的缩进：

```
function func(boolVar) {
  if (boolVar) {
    console.log('True');
  } else {
    console.log('False');
  }
};
```

错误的缩进：

```
function func(boolVar)
{
    if (boolVar)
    {
        console.log('True');
    }
    else
    {
        console.log('False');
    }
};
```

B.2　行宽

尽管现在你的显示器屏幕可能已经很宽了，但为了保证在任何设备上都可以方便地阅读，我们建议把行宽限制为80个字符。

B.3　语句分隔符

JavaScript不仅支持像C语言一样的分号（；）作为语句之间的分隔符，还支持像Python语言那样的换行作为语句之间的界限。我们建议一律使用分号，哪怕一行只有一个语句，也不要省略分号。

正确的语句分隔：

```
var a = 1;
var b = 'world';
var c = function(x) {
  console.log('hello ' + x + a);
};
c(b);
```

错误的语句分隔：

```
var a = 1
var b = 'world'
var c = function(x) {
  console.log('hello ' + x + a)
}
c(b)
```

B.4　变量定义

永远使用 var 定义变量，而不要通过赋值隐式定义变量。因为通过赋值隐式定义的变量总是全局变量，会造成命名空间污染。我们建议绝不使用全局变量，因此要通过 var 把所有变量定义为局部变量。

使用 var 定义变量时，确保每个语句定义一个变量，而不要通过逗号（，）把多个变量隔开。

正确的变量定义格式：

```
var foo;
var bar;
var arr = [40, 'foo'];
var obj = {};
```

错误的变量定义格式：

```
var foo, bar;
var arr = [40, 'foo'],
    obj = {};
```

B.5　变量名和属性名

我们使用小驼峰式命名法（lower camel case）作为所有变量和属性的命名规则，不建议使用任何单字母的变量名。

正确的命名：

```
var yourName = 'BYVoid';
```

错误的命名：

```
var YourName = 'BYVoid';
//或者
var your_name = 'BYVoid';
```

B.6　函数

JavaScript具有函数式编程的特性，因此函数本质上和一般的变量没有区别，对于一般的函数我们同样使用小驼峰式命名法。但对于对象的构造函数名称（或者不严格地说“类”的名称），我们使用大驼峰式命名法（upper camel case），也称为Pascal命名法。

规定函数名与参数表之间规定无空格，参数表和括号（ { 和 } ）之间要有一个空格，并且在同一行。

正确：

```
var someFunction = function() {
  return 'something';
};

function anotherFunction() {
  return 'anything';
}

function DataStructure() {
  this.someProperty = 'initialized';
}
```

错误：

```
var SomeFunction = function()
{
  return 'something';
};

function another_function () {
  return 'anything';
}

function dataStructure() {
  this.someProperty = 'initialized';}
```

B.7　引号

JavaScript中单引号（ ' ）和双引号（ " ）没有任何语义区别，两者都是可用的。我们建议一律统一为单引号，因为JSON、XML都规定了必须是双引号，这样便于无转义地直接引用。

正确的引号用法：

```
console.log('Hello world.');
```

错误的引号用法：

```
console.log("Hello world.");
```

B.8　关联数组的初始化

将 `var = {` 放在一行，下面每行一对键值，保持两空格的缩进，以分号结尾，`};` 最后单独另起一行。对于每对键值，除非键名之中有空格或者有非法字符，否则一律不用引号。

正确：

```
var anObject = {
  name: 'BYVoid',
  website: 'http://www.byvoid.com/',
  'is good': true,
};
```

错误：

```
var anObject = {'name': 'BYVoid',
                website: 'http://www.byvoid.com/'
                , "is good": true};
```

B.9　等号

尽量使用 `===` 而不是 `==` 来判断相等，因为 `==` 包含了隐式类型转换，很多时候可能

与你的预期不同，例如下面错误的例子，`num == literal`的值是`true`。

正确的等号用法：

```
var num = 9;
var literal = '9';
if (num === literal) {
  console.log('Should not be here!!!');
}
```

错误的等号用法：

```
var num = 9;
var literal = '9';
if (num == literal) {
  console.log('Should not be here!!!');
}
```

B.10　命名函数

尽量给构造函数和回调函数命名，这样当你在调试的时候可以看见更清晰的调用栈。

对于回调函数，Node.js的API和各个第三方的模块通常约定回调函数的第一个参数是错误对象`err`，如果没有错误发生，其值为 `undefined`。

正确：

```
req.on('end', function onEnd(err, message) {
  if (err) {
    console.log('Error.');
  }
});

function FooObj() {
  this.foo = 'bar';
}
```

错误：

```
req.on('end', function (message, err) {
  if (err === false) {
    console.log('Error.');
  }
});

var FooObj = function() {
  this.foo = 'bar';
}
```

B.11　对象定义

尽量将所有的成员函数通过原型定义，将属性在构造函数内定义，然后对构造函数使用 new 关键字创建对象。绝对不要把属性作为原型定义，因为当要定义的属性是一个对象的时候，不同实例中的属性会指向同一地址。除非必须，避免把成员函数定义在构造函数内部，否则会有运行时的闭包开销。

正确：

```
function FooObj(bar) {
  //在构造函数中初始化属性
  this.bar = bar;
  this.arr = [1, 2, 3];
}

//使用原型定义成员函数
FooObj.prototype.func = function() {
  console.log(this.arr);
};

var obj1 = new FooObj('obj1');
var obj2 = new FooObj('obj2');
obj1.arr.push(4);
obj1.func(); // [1, 2, 3, 4]
obj2.func(); // [1, 2, 3]
```

错误：

```
function FooObj(bar) {
  this.bar = bar;
  this.func = function() {
    console.log(this.arr);
  };
}

FooObj.prototype.arr = [1, 2, 3];

var obj1 = new FooObj('obj1');
var obj2 = new FooObj('obj2');
obj1.arr.push(4);
obj1.func(); // [1, 2, 3, 4]
obj2.func(); // [1, 2, 3, 4]
```

B.12　继承

首先，避免使用复杂的继承，如多重继承或深层次的继承树。如果的确需要继承，那么

尽量使用Node.js的`util`模块中提供的`inherits`函数。例如我们要让`Foo`继承`EventEmitter`，最好使用以下方式：

```
var util = require('util');
var events = require('events');
function Foo() {
}
util.inherits(Foo, events.EventEmitter);
```

索　　引

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Symbols

“简洁的代码示例，轻快的语言，这本书带你进入同样简明的Node.js世界。如果你想立即使用Node.js进行Web开发，这里提供了绝佳的指导。”

——杨懋，微软亚洲研究院主管研究员

“本书是一本浅显易懂的Node.js入门读物，适合有一定JavaScript基础的开发人员阅读。读过这本书，你就完成了从学习Node.js相关知识，到使用Node.js构建实际Web系统的全过程。难能可贵的是，本书在讲解Node.js的同时，还详细介绍了Web开发领域的通用知识与原理，这些对开发完善健壮的Web应用必不可少。”

——贾超，淘宝网数据产品部资深经理，CNode社区发起者

“在CNode社区企盼将近两年后，第一本中文Node.js图书终于诞生了。跟着BYVoid同学的这本《Node.js开发指南》，你就会走进Node，初探到Node的好和美。”

——田永强（朴灵），淘宝网数据产品部门前端工程师，CNode社区组织者之一

Node.js是一种新兴的开源技术，它将JavaScript从Web浏览器移植到常规的服务器端，使用Chrome的V8虚拟机来解释和执行JavaScript代码，能用于构建高性能、高可扩展的服务器和客户端应用，以实现真正“实时的Web应用”。Node.js在GitHub上吸引了大量开发人员的注意，目前已经有不少可以直接引用的优秀模块。

本书是一本Node.js的入门教程，共分6章，分别讨论了Node.js的背景、安装和配置方法、基本特性、核心模块以及开发实战，让读者对Node.js有一个全面的认识，学会如何用Node.js编程，并了解到事件驱动的异步式I/O的编程模式，同时还可以掌握一些使用JavaScript进行函数式编程的方法。本书非常适合想学习新技术的Web应用开发人员阅读。

图灵社区：www.ituring.com.cn
新浪微博：@图灵教育 @图灵社区
反馈/投稿/推荐信箱：contact@turingbook.com
热线：(010)51095186转604

分类建议 计算机/Web开发
人民邮电出版社网址：www.ptpress.com.cn

ISBN 978-7-115-28399-3

ISBN 978-7-115-28399-3
定价：45.00元